

Specification of a Software Architecture for Capability and Quality-of-Service Negotiations and Session Establishment for Distributed Multimedia Applications

5

The underlying invention generally relates to the field of distributed mobile computing in a mobile wireless networking environment with distributed multimedia applications and systems. More specifically, it is directed to the field of quality negotiation and session establishment for adaptive real-time streaming services running on fixed and mobile devices which support different access technologies in dynamic wireless Internet Protocol (IP) networks, thereby including research and development issues which are especially related to multimedia middleware and resource reservation mechanisms.

A problem that distributed systems will most likely face is how to cope with limited resources at the end systems and in the network, and unstable environment conditions. Mobile users are in fact expected to incur more frequently on the unfortunate case of having their QoS Contracts being no longer supported by the network infrastructure, due to various reasons like wireless link quality degradations, horizontal and/or vertical handovers, limited amount of mobile terminal resources, etc., which shall be referred to as „QoS violations“. By assuming proper resource overprovision in the backbone, it can be expected that QoS violations will most likely originate due to handovers or within the access network, especially in the radio part thereof.

30

Furthermore, mobile multimedia applications dealing with multiple media streams of information being simultaneously exchanged with a multiplicity of peers require an effective and efficient way of negotiating capabilities and handling QoS

requirements, especially in view of the aforementioned unstable environment conditions.

A possible way of coping with unstable environment conditions  
5 is to enable the mobile users' applications to efficiently and timely react to QoS violations. Peers can in fact negotiate off-line various alternative QoS Contracts at different levels of abstraction, so that at the time when a connection is established and whenever QoS violations occur, agreements  
10 on how to most effectively adapt to the mutated conditions can timely be accomplished among the peers.

#### BRIEF DESCRIPTION OF THE PRESENT STATE OF THE ART

15 In the European patent application EP 01 122 366.6, the End-to-End Negotiation Protocol (E2ENP) has already been introduced and described in detail. As the present invention develops further the ideas described in this European patent application, its disclosure is hereby incorporated by reference.  
20

The following table gives a brief overview of recently published articles which describe the concept of QoS and capability negotiation protocols (in alphabetical order):

25

Abbr.	Publication
[Ber]	T. Berners-Lee et al.: „Uniform Resource Identifiers (URI): Generic Syntax“, Networking Working Group, Standards Track RFC 2396.
[Bos1]	L. Bos et al.: „A Framework for End-to-End User Perceived Quality of Service Negotiation“, IETF Internet Draft, work in progress, <draft-bos-mmusic-sdpqos-framework-00.txt>. It describes a process for negotiating QoS with SIP and SDP between two peers. However, it does not describe

Abbr.	Publication
	any architecture for the entity using this process.
[Bos2]	L. Bos et al.: „SDPng Extensions for Quality-of-Service Negotiation“, IETF Internet Draft, work in progress, <draft-bos-mmusic-sdpng-qos-00.txt>. It describes a process for negotiating QoS with SIP and SDP between two peers. However, it does not describe any software architecture for the entity using this process.
[BRAIN]	„Concepts for Service Adaptation, Scalability and QoS Handling on Mobility-Enabled Networks“, IST-1999-10050 BRAN Deliverable 1.2 ( <a href="http://www.ist-brain.org/">http://www.ist-brain.org/</a> ) .
[BRENTA]	A. Kassler et al.: „BRENTA - Supporting Mobility and Quality of Service for Adaptable Multimedia Communication“, in: Proceedings of the IST Mobile Communications Summit 2000, Galway, Ireland, October 2000, pp. 403-408.
[Cam1]	G. Camarillo, W. Marshall, J. Rosenberg: „Integration of Resource Management and SIP“, IETF Internet Draft, work in progress, <draft-ietf-sip-manyfolks-resource-07.txt>. It develops the requirements of an „Offer/Answer Model“ based on SDP, also in the sense of reservation. However, it does not describe any software architecture for the entity performing such specific negotiations.
[Cam2]	G. Camarillo et al.: „Grouping of Media Lines in SDP“ (IETF Internet Draft, work in progress, <draft-ietf-mmusic-fid-04.txt>).
[Gam]	E. Gamma et al.: „Design Patterns - Elements of Reusable Object-Oriented Software“, Addison-Wesley, Reading-Massachusetts (USA), 1994.

Abbr.	Publication
[Gu]	X. Gu, K. Nahrstedt et al.: „An XML-based Quality-of-Service-Enabling Language for the Web“, Project Report of the National Science Foundation, 2001.
[Gue]	T. Guenkova-Luy, A. Kassler, J. Eisl, D. Mandato: „Efficient End-to-End QoS Signaling – Concepts and Features“, IETF Internet Draft, work in progress, <draft-guenkova-mmmusic-e2enp-sdpng-00.txt>.
[Kly1]	G. Klyne: „A Syntax for Describing Media Feature Sets“, IETF RFC 2533.
[Kly2]	G. Klyne: „Identifying Composite Media Features“, IETF RFC 2938. It describes an optimization for the algorithm disclosed in [Kly1] by introducing data handles and hashing.
[Kut1]	D. Kutscher et al.: „Requirements for Session Description and Capability Negotiation“ (IETF Internet Draft, work in progress, <draft-kutscher-mmmusic-sdpng-req-01.txt>).
[Kut2]	D. Kutscher et al.: „Session Description and Capability Negotiation“, IETF Internet Draft, work in progress, <draft-ietf-mmmusic-sdpng-05.txt>.
[Man1]	D. Mandato, A. Kassler, T. Robles, G. Neureiter: „Concepts for Service Adaptation, Scalability and QoS Concepts on Mobility-Enabled Networks“ (IST Global Summit 2001, Barcelona, September 2001, pp. 285-293). This article describes the core concepts of the End-to-End Negotiation Protocol (E2ENP). A similar paper has been presented at PIMRC 2001 in San Diego, October 2001.
[Man2]	D. Mandato, A. Kassler, T. Robles and G. Neureiter: „Handling End-to-End QoS in Mobile Heterogeneous Networking Environments“ (PIMRC 2001,

Abbr.	Publication
	San Diego, 30/9/2001 to 3/10/2001, pp. C-49 to C-54)
[ReqSpec]	„End-to-End Negotiation Protocol“, IST-2000-28584/SO/WP1/PI/I/003/a1, MIND Internal Contribution to WP1, Activity 1.3, Work Item 2.
[Ros1]	J. Rosenberg, H. Schulzrinne et al.: „SIP: Session Initiation Protocol“, IETF Standards Track, Network Working Group, RFC 3261. This document describes the Session Initiation Protocol (SIP), which is an application-layer control (signaling) protocol for creating, modifying and terminating sessions with one or more participants.
[Ros2]	J. Rosenberg, H. Schulzrinne: „An Offer/Answer Model with SDP“, IETF Internet Draft, work in progress, <draft-ietf-mmusic-sdp-offer-answer-02.txt>. This document defines a mechanism by which two entities can make use of SDP to provide a common view of a multimedia session between them.

[Cam1] presents a multi-phase call-setup mechanism that makes network QoS and security establishment a precondition to sessions initiated by the Session Initiation Protocol (SIP) and described by the Session Description Protocol (SDP). Network resources are reserved before the session is started, thereby using existing network resource reservation mechanisms (e.g. RSVP).

[Kly1] presents a format to express media feature sets that represent media handling capabilities. In addition, an algorithm is provided that matches the feature sets. It might be used to determine if the capabilities of a sender and receiver are compatible. In addition, in [Kly2] an abbreviated format for composite media feature sets that use a hash of

the feature representation to describe the composite is disclosed.

In [Ros2], a complete model for a one-to-one capabilities negotiation with SDP is described. However, this model suffers from problems caused by the definition of mutually referred information and information on grouping media streams due to the flat hierarchy structure of the SDP.

[Kut1] describes a set of requirements relevant for a framework for a session description and an end-point capability negotiation in multi-party multimedia conferencing scenarios. Depending on user preferences, system capabilities or other constraints, different configurations can be chosen for the conference. Thereby, the need for a negotiation process among the peers is identified, but not described in order to determine a common set of potential configurations and select one of the common configurations to be used for information exchange. This capability negotiation is used to get a valid session description, compatible with the end system capabilities and user preferences of the potential participants. Different negotiation policies may be used to reflect different conference types. They also identify the need for network resource reservation coupled with session setup. Finally, a proposal is drafted for describing capabilities and providing the negotiation language, but not a protocol. Thereby, the solution proposed in [Kut1] does neither consider the negotiation protocol for determining a common set of QoS configuration nor integrate local, peer and network resource reservation.

The most recent version of this IETF draft, in the following referred to as [Kut2], provides a detailed XML Schema specification and a prototype of the Audio Codec and Real-Time Protocol (RTP) Profiles.

In [Bos1], an end-to-end user-perceived QoS negotiation is described, with the presumption that some middleware and services may strongly be involved in the final decision about 5 the negotiated QoS-information of the end peers. The decision as described may be taken over some standard „contract types“. Although it is mentioned that the signaling and the data path may go different ways through the network, it is suggested that some middleware on the way of the negotiation 10 path may influence the negotiation though in general having nothing to do with the data paths. In case this protocol model is deployed, the network is not transparent. The negotiation process according to [Bos1] is performed at one shot interleaving also with some non-QoS information (e.g. security, network admittance, etc.) without considering protocol 15 modularity and information consistency with respect to QoS. With the model of [Bos1], it is only possible to use a push mode for the negotiation, which may be restrictive for some applications and services. The adaptation paths are only de- 20 grading and fixed.

In the articles [Man1], [Man2], and [Cam2], the possibility of grouping media streams is discussed. However, the authors do not consider criteria for the grouping, the possibility of 25 recursive group building (a group of many groups), and the treatment of real, pseudo-real and non-real-time information media streams that also may be grouped. Besides, [Man1] and [Man2] define negotiation steps that may or may not run at one shot but not during independent negotiation phases. For 30 this reason, they do not meet the requirements for the consistency of the negotiated QoS information during a negotiation phase and after it. Thereby, in [Man1] the core concepts of the E2ENP are disclosed. The treatment of colliding „Economy Principle“ applications is also not considered. Moreover, 35 [Man1] and [Man2] describe the possibility of setting and

managing adaptation paths for the QoS adaptation, which is controlled by a third party component - the QoS Broker. However, the authors do not consider any possibility for the end parties to perform and control the negotiations on their own.

5

## PROBLEMS TO BE SOLVED BY THE INVENTION

Nowadays, adaptive applications and/or middleware (e.g. QoS Brokers) do not comprise any component which is capable of handling QoS pre-negotiation, QoS negotiation and/or QoS re-negotiation, resource reservation and/or release coordination offering a technology-independent Application Programming Interface (API), which masks the type of signaling protocols used for implementing those mechanisms. Instead, adaptive multi-stream multimedia applications and/or middleware have directly to be coded against protocols like H.323, the Session Initiation Protocol (SIP) and/or the Session Description Protocol (SDP, in future SDPng as described in [Kut2]). Furthermore, said SIP- and/or SDP-based applications or middleware directly perform a parsing of the SDP data and have to infer the actions to be taken by examining changes on the SDP data from previously exchanged versions thereof.

Another problem is that adaptive applications and/or middleware (e.g. QoS Brokers) are not able to express the information to be negotiated (e.g. capabilities, application-level QoS Contracts, network-level QoS Contracts, stream-level adaptation paths, and stream-group-level adaptation paths) in an interchangeable format. It is required that heterogeneous applications and/or middleware can easily agree on a reference model, which said applications and/or said middleware can then use for dynamically configuring themselves to orchestrate local, peer, and network resources according to the preferences and profiles of the respective user, policies of the respective systems and network providers in a coordinated

manner among a multiplicity of heterogeneous applications and middleware used by various peers.

In the European patent application EP 01 122 366.6, the overall E2ENP concept, its requirements, and a possible implementation idea thereof is disclosed, however, without detailing any implementation. Any pre-negotiated information is not accomplished in time.

Although the current form of SDPng as described in [Kut2] is structured in a modular way, it does not consider E2ENP aspects and can not be used in a modular way across different SIP messages (or other protocol messages). Capability negotiations based on SDPng use the SDP Offerer/Answerer model described in [Ros2], in which complex multi-phase negotiation processes such as the one proposed in the scope of E2ENP are not explicitly taken into account.

A process capable of considering user profile information as input for the overall QoS negotiation process is neither addressed in the European patent application EP 01 122 366.6 nor in SDPng.

#### OBJECT OF THE UNDERLYING INVENTION

In view of the explanations mentioned above, it is the primary object of the underlying invention to propose a negotiation method and a software architecture supporting capability and QoS negotiation and session establishment combined with resource reservation mechanisms for adaptive real-time services and multimedia applications.

This object is achieved by means of the features of the independent claims. Advantageous features are defined in the sub-

ordinate claims. Further objects and advantages of the invention are apparent in the following detailed description.

#### SUMMARY OF THE UNDERLYING INVENTION

The underlying invention is basically dedicated to the concept and realization of a novel User Agent (UA) for an End-to-End Negotiation Protocol (E2ENP) - a software module encapsulating the end-to-end signaling processes of the E2ENP as described in [ReqSpec] -, which can advantageously be applied to define user profile and terminal capability information in such a way that hierarchical QoS Contract specifications (e.g. compelling correlations across different sets of QoS Contracts for related media streams) can be enforced and used for deriving negotiable information. As a reference implementation of this concept, this invention describes a novel usage of the Session Initiation Protocol (SIP) standardized by the Internet Engineering Task Force (IETF) in conjunction with extensions of the Session Description Protocol - Next Generation (SDPng) specification based on the Extensible Markup Language (XML) in order to implement End-to-End QoS Negotiation Protocol (E2ENP) concepts. More specifically, the hereby proposed model extends the SDPng negotiation mechanisms by defining user profile and terminal capability information which allow to enforce and use hierarchical QoS Contract specifications.

Thereby, said End-to-End Negotiation Protocol (E2ENP) is applied to derive negotiable information, which enables a pre-negotiation, fast negotiation and a fast, dynamic re-negotiation of the end-to-end quality and capabilities for a telecommunication session, for multiple configurations of two or a multiplicity of end peers and/or middleware in a consistent, reliable, and incremental way by enabling the mobile applications to efficiently and timely react to QoS viola-

tions. Furthermore, the invention pertains to the concept and realization of a novel E2ENP User Agent which encapsulates the signaling part of E2ENP. Said E2ENP User Agent expresses the information to be negotiated in an interchangeable format  
5 in such a way that heterogeneous applications can easily agree on a reference model, which can then be used for dynamically configuring Finite State Machines to orchestrate local, peer, and network resources according to the preferences and profiles of the respective user in a coordinated  
10 manner.

According to the underlying invention, the conversion of an interchangeable format between an internal (application- and/or middleware-specific) and an external (SIP User Agent specific) representation is performed by a Parser and a Factory implementation. These software components are also configurable according to the specific needs of the underlying carrier protocol agent, the E2ENP User Agent, and the respective application and/or middleware. In this connection, the specific E2ENP User Agent session identification and the respective application and/or middleware session identification of the applied negotiation processes and sub-processes are uniquely mapped to each other and cached in order to be able to uniquely identify pre-negotiation, negotiation and/or re-negotiation sessions. Thereby, the Parser, the Factory, and the Cache are coupled with the implementations of the respective E2ENP User Agent. It should be noted that the configuration of these four software components - Parser, Factory, Cache, and E2ENP User Agent - and the User Agent of the applied carrier protocol is an application-specific task, which depends on the needs of the respectively employed application and/or middleware component.  
20  
25  
30

## BRIEF DESCRIPTION OF THE DRAWINGS

Further advantages and possible applications of the underlying invention result from the subordinate claims as well as from the following description of the preferred embodiment of the invention, illustrated by the following figures and tables:

Fig. 1 presents a block diagram showing the applied architecture for the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP) according to the underlying invention,

Fig. 2 shows a first implementation example of the applied architecture for the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP) according to one embodiment of the underlying invention using a Java-based SIP stack (JSIP), an SDPng Parser and Factory,

Fig. 3 shows a second implementation example of the applied architecture for the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP) according to one embodiment of the underlying invention using Sun's Java Remote Method Invocation (RMI),

Fig. 4 shows a third implementation example of the applied architecture for the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP) according to one embodiment of the underlying invention using the socket-based User Datagram Protocol (UDP) or Transmission Control Protocol (TCP),

Fig. 5 exhibits a UML class diagram showing the org::mind::e2enp package,

Fig. 5<sup>bis</sup> shows the E2ENP Management API between the E2ENP UA Factory and the Management Entities,

Fig. 6 presents a document showing examples for the syntax of the E2ENP Universal Resource Identifier (URI), thereby

using the Augmented Backus-Naur Form (ABNF),  
Fig. 7 outlines a first message sequence chart (MSC) showing the pre-negotiation procedure enabled by the User Agent (UA) of the proposed End-to-End Negotiation Protocol (E2ENP),  
Fig. 8 outlines a second message sequence chart (MSC) showing the session establishment with a QoS negotiation and resource reservation coordination enabled by the User Agent (UA) of the proposed End-to-End Negotiation Protocol (E2ENP),  
Fig. 9 exhibits a UML class diagram showing the org::mind::e2enp::Cache sub-package,  
Fig. 10 presents a diagram showing the top-level view of an Extensible Markup Language (XML) description used for the End-to-End Negotiation Protocol (E2ENP),  
Fig. 11 presents a diagram showing the XML substitution groups used for the End-to-End Negotiation Protocol (E2ENP),  
Fig. 12 presents a diagram showing the XML purpose element used for the End-to-End Negotiation Protocol (E2ENP),  
Fig. 13 presents a diagram showing the XML qosdef element used for the End-to-End Negotiation Protocol (E2ENP),  
Fig. 14 presents a diagram showing the XML qoscfg element used for the End-to-End Negotiation Protocol (E2ENP),  
Fig. 15 exhibits a UML message sequence chart (MSC) showing the interaction between the E2ENP User Agent (E2ENP UA) according to the underlying invention and the applied Parser and Cache,  
Fig. 17 exhibits a UML class diagram showing the general structure of a Document Object Model (DOM) tree,  
Fig. 18 exhibits a UML class diagram showing a structural overview of the Parser implementation using a visitor design pattern,  
Fig. 19 presents a UML message sequence chart (MSC) showing the interaction between the E2ENP User Agent (E2ENP UA) according to the underlying invention and the applied Fac-

tory and Cache,

Fig. 20 presents a UML class diagram showing a structural overview of the Factory implementation,

Fig. 21 exhibits a UML class diagram showing the org::mind::sip::sipApi package,

Fig. 22 exhibits a UML class diagram showing the org::mind::sip::sipApi::userAgent package,

Fig. 23 exhibits a UML class diagram showing the org::mind::sip::sipApi::registrar package,

Fig. 24 exhibits a UML class diagram showing the org::mind::sip::sipApi::management package,

Fig. 25 exhibits a UML class diagram showing the org::mind::sip::sipApi::time package,

Fig. 26 exhibits a UML message sequence chart (MSC) showing the configuration of the Service Provider and the binding of the Service User with said Service Provider,

Fig. 27 exhibits a UML message sequence chart (MSC) showing the connection establishment between the User Agents of a client and a server,

Fig. 28 exhibits a UML message sequence chart (MSC) showing the OPTIONS method used for the End-to-End Negotiation Protocol (E2ENP),

Fig. 29 exhibits a UML message sequence chart (MSC) showing the connection release between the User Agents of a client and a server,

Fig. 30 shows a first state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the root state,

Fig. 31 shows a second state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the NegOfferer sub-state,

Fig. 32 shows a third state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the NegAnswerer sub-state,

Fig. 33 shows a fourth state transition diagram for a nested Fi-

nite State Machine (FSM) showing the mutex-related procedures executed in the ReNegOfferer sub-state,

Fig. 34 shows a fifth state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the ReNegAnswerer sub-state,

Fig. 35 shows a sixth state transition diagram for the „\_Resv-Mtx“ Finite State Machine (FSM) for allowing multiple requests to seize the mutex in a coordinated manner, based on their priority,

Fig. 36 shows a seventh state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the WaitForCoordDone sub-state,

Fig. 37 exhibits a UML message sequence chart (MSC) showing the pre-negotiation procedure needed for the correlation of the Application Programming Interface (API) of the E2ENP User Agent (E2ENP UA) and the generic Application Programming Interface (API) of the SIP User Agent (SIP UA), thereby using the above-mentioned Finite State Machine (FSM) of the E2ENP User Agent (E2ENP UA),

Fig. 38 exhibits a UML message sequence chart (MSC) showing the session establishment with the QoS negotiation procedure and the resource reservation coordination needed for the correlation of the Application Programming Interface (API) of the E2ENP User Agent (E2ENP UA) and the generic Application Programming Interface (API) of the SIP User Agent (SIP UA), thereby using the above-mentioned Finite State Machine (FSM) of the E2ENP User Agent (E2ENP UA),

Tab. 1 shows a list of primitives implemented by the Service Provider, based on a Java implementation of the Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 2 shows a list of primitives to be implemented by the Service User, based on a Java implementation of the Application Programming Interface (API) applied to the

User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 3 shows a list of primitives to be implemented by the Service User acting as an Offerer, based on a Java implementation of the Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 4 shows a list of primitives to be implemented by the Service User acting as an Answerer, based on a Java implementation of the Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 5 shows the methods provided by the Application Programming Interface (API) of the first Cache level,

Tab. 6 shows the methods provided by the Application Programming Interface (API) of the second Cache level,

Tab. 7 shows a list of primitives which have to be implemented by the Application Programming Interface (API) of the Parser,

Tab. 8 shows a list of primitives which have to be implemented for generating a specific Parser instance,

Tab. 9 shows a list of primitives which have to be implemented by the Application Programming Interface (API) of the Factory,

Tab. 10 shows a list of primitives which have to be implemented for generating a specific Factory instance,

Tab. 11 shows the methods provided by the well-known Factory-Factory class E2ENPContentHandlerFactory,

Tab. 12 shows a list of primitives implemented by the Service Provider of the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP), based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 13 shows a list of client-side-specific primitives imple-

mented by the Service User, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 14 shows a list of server-side-specific primitives implemented by the Service User, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 15 shows a list of both client-side- and server-side-specific primitives implemented by the Service User, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 16 shows a list of primitives implemented by the Service Provider, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 17 shows a list of primitives implemented by the Service User, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP),

Tab. 18 is a state transition table showing a survey of the applied exception events, and

Tab. 19 shows the applied timers applied to the End-to-End Negotiation Protocol (E2ENP) according to the underlying invention.

## DETAILED DESCRIPTION OF THE UNDERLYING INVENTION

According to one embodiment of the underlying invention, an  
5 E2ENP UA 128 supports the following platforms: Win32 and  
Linux. Quality, robustness, maintainability, extensibility,  
flexibility and efficiency are the challenges to the archi-  
tecture and its solutions.

10 – Quality includes testability and verification of the system  
already during development. Robustness is the ability of  
the system to stay stable and available also in error  
situations. Both aims directly affect the availability and  
acceptance of the system.

15 – Maintainability and extensibility include all costs of the  
system in later phases (concerning both error correction  
and extensions). These costs must be minimized.

20 – Flexibility also addresses the point that changes should be  
applied as cost-effective as possible.  
  
– Efficiency addresses minimization of resource usage (con-  
cerning memory and processor).

25 For the architecture, the following basic conceptual and  
technical requirements can be derived:

- ◆ definition of design components with well chosen and de-  
30 fined responsibility and features;
- ◆ small dependencies between design components (loose cou-  
pling), so that changes have only a limited effect;
- ◆ minimum resource consumption, especially concerning the  
applied Random Access Memories (RAMs);

- ◆ a strict separation between interface and implementation (a change of the implementation shall not affect the interface);
- ◆ definition of restrictions, if this simplifies the realization concept – however, restrictions and their effects must be negotiated with the stakeholders.

5 The E2ENP UA 128 depicted in Fig. 1 is designed as a software component which communicates with the environment through the  
10 following five APIs:

- E2ENP Management API 101a (IF1): This API represents the interface between the E2ENP UA 128 and any entities 102 managing it, in the sense of configuration, monitoring, and  
15 control.
- E2ENP UA API 101b (IF2): This API represents the interface between the E2ENP UA 128 and any application 130 or middleware using the services offered by said E2ENP UA 128. The  
20 information exchanged through this API can be modeled as an object structure or as an XML-based document, to be accordingly interpreted by applications 130 or middleware 130 placed on top of the E2ENP UA 128.
- 25 • SIP UA Generic API 101c (IF3): This API represents the interface between the E2ENP UA 128 and the session-layer protocol used for implementing the E2ENP-specification. The name of this API clearly indicates that the Session Initiation Protocol (SIP) described in [Ros1] is used as a reference session-layer protocol. However, with minor modifications, this API can easily be generalized in a future design review, so as to remove the E2ENP UA dependency on the SIP. Nevertheless, by using this design it would be possible to have a complete Java Remote Method Invocation (RMI)  
30 implementation of the SIP UA Generic API 101c (IF3).
- 35

- Parser API 101d (IF4): This API represents the interface between the E2ENP UA 128 and any Parser implementation 112 used for decoding the E2ENP session description.

5

- Factory API 101e (IF5): This API represents the interface between the E2ENP UA 128 and any Factory tool 114 used for encoding the E2ENP session description.

10 Internally, the E2ENP UA 128 is composed of different Finite State Machines 106 (FSMs) handling the coordination of the procedures associated with the aforementioned interfaces, and their interrelations. More specifically, the E2ENP UA 128 should feature a client FSM 106 and a server FSM 106 for in-

15 independently and simultaneously handling, respectively, the initiation of an E2ENP (pre-)negotiation and/or session establishment, and the response to such procedures. This means mapping primitives of the interface 101b (IF2) to primitives of the interface 101c (IF3), and vice versa. The mapping of

20 the primitives concerns both object and object identifiers mapping. The mapped objects and the identifiers are stored in a respective Cache 104 at the E2ENP UA 128 and/or at the application 130 above IF2.

25 During this mapping process, the E2ENP UA FSMs 106 should also be able to access the interfaces 101d (IF4) and 101e (IF5) for handling, respectively, E2ENP session description decoding and encoding.

30 The Parser and Factory implementations 112 and 114, respectively, though essential entities for implementing the E2ENP concept, are designed as separated software components, insofar as they can be implemented in various ways (based on SDP, SDPng, SDPng dialects, or any other session description languages of choice).

In the following subsection, the basic problem domains of the system shall be described and how they are solved within the E2ENP UA 128. Each problem domain is discussed individually  
5 and across the system. It refers to other problem domains if there is any interference with them or if a part of the problem domain is discussed more detailed in an extra section.

The E2ENP UA 128 is designed according to the object-oriented paradigm, and a Java-based prototype based on such design is hereby described. It allows using various types of XML parsing tools (through IF4) and various session level protocols (through IF3) to transport E2ENP primitives. The hereby-described prototype based on the aforementioned E2ENP UA design uses SIP as described in [Ros1] as session-layer protocols. Various SIP implementations are supported by the E2ENP UA design, thanks to the abstract nature of IF3.  
10  
15

In the scope of the underlying invention, both the E2ENP UA  
20 FSM 106 and the SIP UA Generic API (IF3) are explicitly designed to support SIP. It is foreseen that in a future design review these dependencies will be removed by making IF3 even more abstract. The E2ENP UA 128 will also be able to simultaneously support multiple users of multiple applications  
25 130 using multiple instances of the E2ENP UA services.

The E2ENP UA 128 supports three types of object categories:  
30 identifiers (IDs), description objects (object graphs or a formal description of an object graph, like XML) and event objects. These categories correspond to the objects passed to the agent through IF2, IF3, IF4, and IF5 in both directions.

- The IDs are needed to uniquely identify the session description objects. There are two types of identifiers,  
35 which are also objects:

- E2ENP internal object IDs, corresponding to the objects processed within the E2ENP UA FSMs 106,
- application IDs pointing also at the respective E2ENP UA 5 FSM 106 objects, which are used by an application 130 to steer negotiation events.

10

- The session description objects are either object graphs used by the E2ENP UA FSM processing or external descriptions used by the Parser implementation 112, the Factory implementation 114, and the E2ENP UA 128 by a negotiation process. All these session description objects can persistently be saved by the application 130 (above IF2) using them. The management (leasing descriptions, refreshing of leased descriptions, etc.) of the saved objects is an application-specific task.

15

- The event objects correspond to application negotiation events, which affect the E2ENP UA FSM 106 functionality by triggering adaptation sequences.

20

The IDs, the E2ENP UA FSM object graphs and the event objects are Java objects. The external protocol representations can be SDP, SDPng, SDPng dialects, or any other session description language objects of choice. The implementation of the 25 Java objects passed through interface IF2 depend on the interface definition of IF2. The objects passed through the interfaces IF4 and IF5 are either E2ENP UA FSM 106 object representations or external protocol representations. The objects passed through IF3 are external protocol representations. All the external representations of the session 30 description objects passed through IF3, IF4 and IF5 should implement java.io.Serializable in order to be compatible with the SIP Surrogate RMI implementation.

The internal representations of the session description objects passed over the IF2, IF3, IF4 and IF5 shall be typed as java.lang.Object and should be cast into the expected type by  
5 the E2ENP UA 128, the Parser 112, and the Factory 114. This requirement is necessary to keep the SIP UA 110 and the implementations of the Parser 112 and the Factory 114 modular by leveraging generic APIs.

10 The following section contains a brief survey of the applied implementations of the Parser 112 and the Factory 114.

Since the negotiation process involves at least two entities interconnected by a network, it is necessary to provide mapping functionality from system-internal representation to a transport representation over the network and back. The mechanisms that provide this mapping are called Parser API 101d (in the following referred to just as Parser 112) and Factory API 101e (referred to as Factory 114), where the  
15 Parser 112 takes an object encoded in network representation as input and generates a system-internal representation of said object. In contrast, the Factory 114 generates a network representation for a given system-internal representation.  
20

25 The need for said Parser API 101d, said Factory API 101e and the functionality defined by them arises from the facts that the system-internal representation may not be appropriate for a transport representation that can be sent over networks and the transport representation may not be adequate for a system-internal representation.  
30

Additionally, introducing these components into the system decouples the higher application layer from the lower transport-oriented layers. As a consequence, the actually used

transport protocol is of no concern to the upper application layers and should ideally be pluggable.

The use of the Extensible Markup Language (XML) as transport syntax is motivated by the widespread adoption of XML as an industry standard, and it also allows for the incorporation of external features (SDPng libraries and profiles) and provides compatibility to related work (SDPng schema). Another feature of XML is its simple extensibility.

As described above, the use of different carrier protocols requires that the Parser 112 and the Factory 114 are configurable by the same means as the protocol, which means that in case the protocol is configurable at runtime, said Parser 112 and said Factory 114 might also have to be changed while the system is running. This implies the specification of an abstract interface, since the actual implementations depend on the specific underlying carrier protocols.

Concrete Factory 114 implementations for different carrier protocols will produce different results. For example, an RMI factory may just be a dummy implementation. It is thus possible to transfer Java objects, e.g. the objects „as is”, directly over a network connection by simply using Java Remote Method Invocation (RMI).

If SIP is used as a carrier protocol, however, an external representation (more specifically a protocol data unit or PDU) to be sent over the network has to be generated. Possibilities include but are not limited to some proprietary text format similar to SDP, for example, or a more structured approach using XML, as e.g. SDPng does.

The API definitions for the Parser API 101d and the Factory API 101e are un-typed in order to support loose coupling of

the E2ENP UA 128 and the Parser 112 as well as the Factory 114. The transport representation as described above is implemented by just using the java.io.Serializable interface, which is inherently un-typed. The system-internal representation is chosen to be the represented as java.lang.Object. The matching between the actual implementations of the Parser 112 and the Factory 114 and the object representation used in the E2ENP UA 128 have to be established by configuration.

10 The mechanism is implemented by the following two software components:

- ◆ Parser API 101d: Primitives of this API should accept as parameters any object implementing the java.io.Serializable interface. The result of the parsing process yields an object representation according to IF2, which will abstractly be typed as java.lang.Object.

- ◆ Factory API 101e: Primitives of this API should accept as parameter any object representation according to IF2, which will be abstractly typed as java.lang.Object and generate a representation conforming to the java.io.Serializable interface.

25 Additionally, in order to achieve configurability, the use of a „factory design pattern“ for creating Parser 112 and Factory 114 instances is a possible solution. For this purpose, the factory method is parameterized with the transport protocol (the distinguishing feature of different Parser  
30 112/Factory 114 implementations). If dynamic support for different carrier protocols is needed, the factory classes have to provide a registration procedure for new protocols.

In the next section, the use of SIP surrogates shall briefly be described.

The applied SIP UA Generic API 101c (IF3) requires the availability of a SIP UA implementation which supports such an API. Similarly, the Parser API 101d (IF4) and the Factory API 101e (F5) presume the availability of parser and factory implementations compatible with the UA implementation 110, respectively. In order to allow validating and rapidly prototyping the E2ENP model, IF3, IF4, and IF5 have been designed in such a way that surrogates of real SIP UA implementations and of compatible Parser 112 and Factory 114 implementations can be used as handy alternatives for experimental purposes.

By using a legacy implementation (already provided code/API and/or software, possibly from an external implementer) of the SIP UA 110 for testing and/or updating the E2ENP UA 128, it might be sometimes difficult to establish the connection between the E2ENP UA 128 and the SIP UA 110, as the interface provided by the legacy SIP implementation may not match directly the interface 101c required by the E2ENP UA. In general, using a SIP UA 110 to provide network connectivity for the E2ENP UA 128 is not recommended whenever testing new E2ENP UA 128 features, due to possible conceptual contradictions between the protocol agents (SIP and E2ENP). These contradictions may result in specification changes of the carrier protocol - SIP, that is why for the tests of the E2ENP UA 128 the usage of a SIP emulator (which can easily adopt conceptual changes) is preferable.

In order to carry out a SIP emulation, SIP surrogates (e.g. Remote Procedure Call mechanisms, RPCs) can be employed. When E2ENP session description payloads have to be tested without necessarily using „full-blown“ Parser 112 and Factory 114 im-

plementations, solutions coupled with the chosen SIP surrogate can also be applied.

As mentioned above, a prototype E2ENP UA 128 implementation based on the hereby-presented architectural model has been selected for demonstrating the E2ENP concept. Since this implementation is based on Java, an RPC-based, built-in Java RMI mechanism has been selected as SIP surrogate.

Since RMI takes care of marshalling or demarshalling the SIP UA Generic API 101c (IF3) primitives payload directly, all the parameters passed to the API have to implement the java.io.Serializable interface.

In order to test E2ENP session description payloads without necessarily using „full-blown“ Parser 112 and Factory 114 implementations (for the same reasons set forth above for the use of SIP surrogates), passing such content through the IF3 as a string might be not necessary as RMI can process objects. Rather, a whole object structure representing a given E2ENP session description payload might be exchanged directly through the RMI marshalling or demarshalling mechanism. Therefore, the prototype E2ENP UA 128 implementation based on the hereby-presented architectural model has been designed in such a way that the IF3 primitives shall accept as parameters any object implementing the java.io.Serializable interface, and not just instances of the java.lang.String class.

Said mechanism is implemented by the following software components:

- ◆ SIP UA Generic API 101c: Primitives of this API shall accept as parameters any object implementing the java.io.Serializable interface, and not just instances of the java.lang.String class. A SIP UA 110 implementation shall inter-

5  
pret the passed objects as instances of the java.lang.String class via casting mechanism. The RMI surrogate SIP implementation shall directly serialize or deserialize the aforementioned objects „as is“.

- ◆ Parser API 101d: Primitives of this API shall accept as parameters any object implementing the java.io.Serializable interface.
- 10 ◆ Factory API 101e: Primitives of this API shall accept as parameters any object implementing the java.io.Serializable interface.

15 Fig. 2 shows an implementation of the architectural model as mentioned above, based on a specific SIP UA 110 implementation (e.g. a Java based SIP stack - JSIP), and on Factory 114 and Parser 112 implementations handling a modified version of SDPng.

20 Fig. 3 shows an implementation of the architectural model mentioned above, based on an RMI SIP surrogate: In this case, the Factory 114b and Parser 112b implementations are simply dummies.

25 The Dummy ParserFactory 118b and the Dummy FactoryFactory 120b respectively create the „Dummy Parser“ runtime instance 112b and the „Dummy Factory“ runtime instance 114b.

30 Though indicated as „dummy“, still, said Parser 112b and said Factory 114b could check and/or enforce that the objects used in the object graph are typed as „java.io.Serializable“.

35 Fig. 4 shows an implementation of the architectural model mentioned above, using a socket-based SIP surrogate: in this case, the Factory 114c and Parser 112c implementations are

simply dummies. Nevertheless, one could also use the SDPng Parser 112a and SDPng Factory 114a to send XML based strings across the sockets using a proper Adapter 126c. The primitives are manually mapped into messages to be sent over sockets, thus implementing a proprietary Remote Procedure Call (RPC) mechanism.

10 The Dummy FactoryFactory 120c and Dummy ParserFactory 118c create respectively the „Dummy Parser“ runtime instance 112c and the „Dummy Factory“ runtime instance 114c.

15 Though indicated as „dummy“, still, said Parser 112c and said Factory 114c could check and/or enforce that the objects used in the object graph are typed as „java.io.Serializable“.

20 In the following subsections, the software components applied within the scope of the E2ENP UA 128 according to the underlying invention shall briefly be described from the viewpoint of the component. For each software component, its requirements, offered services and constraints are described and also its relations to other software components. This description serves as the basis for the detailed design.

25 Fig. 1 shows all software components and their usage relations. A software component is assigned to a layer and may have relations to components on the same or a lower layer but not to software components on a higher layer. The layers do not have the strict meaning of encapsulating a lower layer.

30 Accordingly, the entity using the services offered by a given API will be referred to as Service User, whereas the entity implementing the services of said API will be referred to as Service Provider.

The E2ENP UA API 101b (the aforementioned IF2) exposes the E2ENP UA functionality to Service Users like QoS-aware applications 130 and/or QoS Brokers 130, according to the BRENTA model as described in [BRENTA]. This component realizes the 5 specification of a generic API by defining a set of interfaces and complex data types exchanged through said API.

Thereby, said E2ENP UA API 101b allows multiple instances of Service Users to simultaneously access the E2ENP UA 128 functionality. More specifically, the E2ENP UA 128 offers the 10 following services as described in [ReqSpec]:

1. pre-negotiation of a multiplicity of alternative capabilities and/or QoS Contracts (at both application and 15 network level);
2. management of leased pre-negotiated information;
3. session establishment with negotiation of a multiplicity of alternative capabilities and/or QoS Contracts (at both application and network level), potentially referencing pre-negotiated information;
4. same as point 2, but with additional local, peer, and 20 network resource reservation coordination;
5. re-negotiation, similar to points 2 and 3;
6. session release, with potential reservation release coordination.

In order to expose these services, the E2ENP UA API 101b shall expose, respectively, the following primitives:

- 30 1. pre-negotiation;
2. leaseRenewal, which can be used for refreshing or terminating a lease;
3. negotiation, parameterized to indicate whether resource reservation coordination is required. If resource reservation coordination is required, the E2ENP UA 128 allows 35

the Service User at the Offerer side to book the mutually exclusive access to the resource reservation phase via the bookNegotiation and/or bookRenegotiation primitive;

- 5     4. re-negotiation: If resource reservation coordination has been specified as required in the original negotiation phase, the E2ENP UA 128 allows the Service User at the Offerer side to book the mutually exclusive access to the resource reservation phase via the bookRenegotiation primitive;
- 10    5. release.

The E2ENP UA API 101b has been designed based on the following design principles:

- 15    1. Primitives are classified into Request (Req), Indication (Ind), Response (Rsp), and Confirmation (Cfm) primitives, according to the ISO/OSI model. Request (Req) and Response (Rsp) primitives are invoked by the client application 130 or middleware 130 implementation (also known as Service User), in order to respectively initiate or respond to a particular message exchange. (In some cases the message exchange may involve a set of different messages being exchanged between peers, thanks to the abstraction offered by the given E2ENP UA implementation, whereas in other cases the message exchange is limited to the given message and its acknowledgment.) Indication (Ind) and Confirmation (Cfm) primitives are invoked by the E2ENP UA 128 (also known as Service Provider) to the registered client-side of the Service User 130, in order to respectively indicate the arrival of a particular message exchange or to confirm the conclusion of a given message exchange. (Again, in some cases the message exchange may involve a set of different messages being exchanged between peers, thanks to the abstraction offered by the given Service Provider,

whereas in other cases the message exchange is limited to the given message and its acknowledgment.) The client-side of the Service User 130 must implement these primitives. To a request primitive generated by the peer initiating the given protocol message exchange, corresponds an Indication (Ind) primitive at the target peer. To a Response (Rsp) primitive generated by the target peer, responding to the given protocol message exchange, corresponds a Confirmation (Cfm) primitive at the initiating peer.

10

- 5 2. In order to distinguish different Service Users 130, the E2ENP UA 128 exposes at interface 101b (IF2) one Service Access Point (SAP) per registered Service User 130. Thereby, any given Service User 130 can use more than one SAP at said interface 101b (IF2) in a one-to-many relationship.
- 15 3. Users can only be registered with one Service User 130 at a time, which means that each user will be associated with at most one SAP at said interface 101b (IF2).
- 20 4. In order to simultaneously and independently use different session layer protocols 110 (e.g. various SIP implementations and/or RMIs), the E2ENP UA 128 exposes at interface 101c (IF3) one SAP per registered session layer protocol.
- 25 5. In order to properly associate Service Users 130 with underlying session layer protocols, the E2ENP UA 128 exposes at interface 101a (IF1) proper configuration mechanisms, which allow associating SAPs at interface 101b (IF2) with SAPs at interface 101c (IF3) in a one-to-one relationship. In this way, user using a given type of application 130 will automatically and transparently be using the associated session layer protocol as configured.

6. When configuring a SAP at interface 101c (IF3), the binding of the E2ENP UA 128 to the session-layer protocol 110 is done automatically. With the given SIP UA 110 Generic API at interface 101c (IF3) the binding in the opposite direction is done when registering a user with the E2ENP UA 128, and this leads to a registration of the user to the configured session layer protocol 110.

5  
10 7. The bidirectional binding of the Service User 130 to the E2ENP UA 128 is done by means of the bindReq and bindCfm primitives, where one of the parameters is a special identifier, and the SAP ID (spId) uniquely identifies one of the configured SAPs at interface 101b (IF2).

15 8. Any given user of any given Service User 130 registers himself/herself on the proper Upper SAP by means of the registrationReq and registrationCfm primitives, where one of the parameters is the aforementioned spId, which uniquely identifies the SAP of choice at interface 101b  
20 (IF2).

25 9. In order to select the proper Upper SAP when locally initiating any E2ENP session, the following primitives now feature a new parameter „String user“ indicating the locally registered user, which uniquely identifies the SAP at interface 101b (IF2) said user is registered to:

- renewLeaseReq and
- bookNegotiationReq

30 10. To select the proper Upper SAP when locally initiating any E2ENP session, the already existing „String form“ parameter of the following primitives indicates the locally registered user, which uniquely identifies the SAP at interface 101b (IF2) said user is registered to:  
35

- preNegotiationReq and
- negotiationReq

5    11. The instances of the application or middleware FSM 130  
and of the E2ENP UA FSM 106 are uniquely identified, re-  
spectively, by the serviceUserId and serviceProviderId.

10    12. A simplified version of the „observer design pattern“ as  
described in [Gam] (implemented in Java with a derivation  
of the Java Beans event model) offers the mechanism al-  
lowing the Service User 130 to register (via a primitive  
called „Bind Request“) for Indication (Ind) and/or Con-  
firmation (Cfm) primitive callbacks.

15    13. In order to allow supporting E2ENP signaling extensibil-  
ity, most of the primitives allow passing a payload: To-  
day, some of these primitives are in fact not designed to  
carry payloads, but there is a chance that the E2ENP  
20    specification might be reviewed. Thereby, some additional  
E2ENP information piggybacking could be required.

25    14. In order to allow different types of Service Users 130  
accessing the services of the E2ENP UA 110, the payloads  
are typed as the Java root class, java.lang.Object:. In  
this way, applications 130 or middleware may use either  
internal custom representations of the information to be  
exchanged via the E2ENP (e.g. an XML-based representa-  
30    tion), or resort to the default information structure de-  
scribed above.

Thereby, said E2ENP UA API 101b depends on the respective ap-  
plication 130 and/or middleware 130 using the services of  
said E2ENP UA 128, the applied E2ENP UA FSM 106, concerning  
35    the implementation of the Request (Req) and Response (Rsp)

primitives, and the applied Factory 114 and Parser 112: The chosen implementation of these components dictates the type of representation used for the information exchanged through this API.

5

The E2ENP UA API 101b has been designed with the aid of the object-oriented paradigm, and is thus hereby described using the Unified Modeling Language (UML). The information to be exchanged via the E2ENP is described above. The component is 10 composed of a basic package, the org::mind::e2enp as depicted in Fig. 5.

To this basic package belong the following interfaces:

15 – The Provider 504 represents the interface that an E2ENP UA Service Provider implementation conforming to the herewith specification, shall support for implementing Request (Req) and Response (Rsp) primitives. Tab. 1 lists the primitives exposed by the Provider interface 504.

20

– The E2ENPUserAgentListener 502 generalizes all the interfaces that Service Users shall implement for intercepting Indication and/or Confirmation (Cfm) primitives generated by the E2ENP UA 128.

25

Tab. 2 lists the primitives exposed by the E2ENPUserAgent-Listener interface 502.

30 The E2ENPUserAgentListener interface 502 is specialized in the OffererListener interface 506 and the AnswererListener interface 508. The former is specially designed for the application- or middleware-initiating E2ENP procedures; the latter is specially designed for applications 130 or middleware 130 responding to said E2ENP procedures.

35

Tab. 3 shows the primitives exposed by the OffererListener interface 506, and Tab. 4 lists the primitives exposed by the AnswererListener interface 508.

5 The „to/from“-parameters passed on by the pre-negotiation and  
negotiation primitives are addresses identifying E2ENP users  
(respectively, the Offerer and the Answerer). The E2ENP UA  
128 maps these addresses to the specific syntax used by the  
given session-layer protocol that is used for piggybacking  
10 E2ENP information: in the case of this writing, SIP. The  
E2ENP addresses shall therefore be independent of the spe-  
cific session-layer protocol, and possibly of the transport  
protocol used underneath by the E2ENP UA 128. To this extent,  
a new syntax, which has specially been designed for the  
15 E2ENP, shall herewith be proposed.

Fig. 5<sup>bis</sup> shows the E2ENP Management API 101a (IF1) between  
the E2ENP UA Factory 108 and the Management Entities 102, de-  
signed by means of the object-oriented paradigm. Thereby, the  
20 E2ENP Management API 101a (IF1) is composed of a basic pack-  
age, to which the following components belong: the Manager-  
Provider interface 510 and the ManagerListener interface 512.  
For the configuration of the Parser implementation 112 and  
the Factory implementation 114 via said interface IF1, the  
25 classes ConfigurationRequest 514 and ParserFactoryConfigura-  
tion 516 are used.

Fig. 6 presents a formal description of such a Universal Re-  
source Identifier (URI) syntax by using the Augmented Backus-  
30 Naur Form (ABNF). Thereby, said specification is similar to  
the SIP URI syntax specification described in [Ros1], but de-  
liberately does not indicate any transport protocol parame-  
ter, except for the IP address and/or port number. As an ex-  
ample of E2ENP URI, „e2enp://dave:vG\$1809@acme.com“ repre-  
35 sents a user named „Dave“ at the „acme“ organization, using

password „vG\$1809“. The „//“ indication is a form of a separator for default user cases, when the user is not indicated in the address. The example „e2enp://:xyz%45637@acme.com“ shows a default user (which is not indicated) with a password „xyz%45637“.

The full or partial usage of the E2ENP address components depends on the domain model (proxying) and the address resolution mechanisms, which are mainly part of the respectively applied session layer protocol (e.g. SIP). The E2ENP UA 128 should use at least the user (default user) name and the domain name to identify the communicating partners, if the session layer protocol is already offering some proxying and/or redirection mechanisms to identify the network paths.

In the following subsections, the procedures executed by the E2ENP UA API 101b shall be described by means of a set of UML Message Sequence Charts (MSCs).

Fig. 7 depicts the simple pre-negotiation scenario in terms of an exchange of primitives through the E2ENP API. With respect to these primitives, specific transport protocol messages are exchanged between the OffererServiceProvider 704 and the AnswererServiceProvider 706, but they are not indicated in Fig. 7 for the sake of generality.

Fig. 8 depicts the MSC for a simple session establishment scenario, wherein QoS negotiation and resource reservation coordination processes are highlighted. In the presented scenario, for the sake of simplicity, the confirmation of resource reservation completion from the Offerer side arrives before the same happens at the Answerer side. Likewise, this simple scenario assumes for the sake of simplicity that the resource reservation process succeeds in reserving exactly the amount of resource, which has originally been requested.

The re-negotiation procedure is similar to the negotiation procedure, except that the names of the primitives have been changed as follows:

Old Name	New name
bookNegotiation<type>	bookReNegotiation<type>
negotiation<type>	reNegotiation<type>
negotiationStatus<type>	reNegotiationStatus<type>

The Cache 104 is a database for saving different object identifiers. In order to decouple the Service User identification scheme from the one specified for the E2ENP [ReqSpec], an E2ENP Cache 104 is used for:

- ◆ saving and retrieving E2ENP session identifiers specified in [ReqSpec],
- ◆ saving and retrieving the E2ENP session identifiers handled by E2ENP UA FSM 106 (serviceProviderIds), and
- ◆ saving and retrieving E2ENP UA API Service Users' session identifiers (serviceUserIds) .

5 The Cache 104 should be searchable for different identifiers and following different criteria set by the E2ENP UA 128. The E2ENP UA 128 is responsible for the Cache management.

10 The cached objects correspond to the identifier object category. The Cache 104 maintains two types of data:

15 ◆ Pre-cached data: The E2ENP UA pre-caches information (indexed by serviceProviderId as primary key, and serviceUserId and E2ENP session identifiers specified in [ReqSpec] as secondary keys) during the execution of a given E2ENP pre-negotiation or negotiation procedure. Once the given procedure has been successfully completed, the pre-cached data remains cached, so that E2ENP UA API 101b Service Users have reference to such data during a later E2ENP session as described in [ReqSpec].

25 ◆ Cached data: This information can be used later, during a new E2ENP pre-negotiation or negotiation procedure, as a reference to data, which has already been (pre-)negotiated among the peers. For each of said E2ENP sessions, the Cache 104 stores the association between the given ServiceUserId and the corresponding E2ENP session identifier described in [ReqSpec]. Each entry in this Cache 104 is leased between peers and should be refreshed over time as described in [ReqSpec]. The Service Users 130 themselves 30 are responsible for managing their own leases and (pre-) 35

negotiated information. The Cache 104 simply stores non-persistently the *correlations* among previous E2ENP sessions.

5 The E2ENP Cache 104 is a database used for managing the negotiation object IDs. The E2ENP UA FSM 106 requests the Cache 104 services. In an embodiment of this invention, the Cache 104 is designed as a set of java.util.Hashtable objects referencing each other. An alternative implementation based upon  
10 tupel-space technology is left for further study.

In order to keep the E2ENP UA 128 implementation simple, the Cache API 103 between the Cache 104 and the E2ENP UA FSM 106 represents a synchronous interface.

15 In the implemented version of the underlying invention, a two-stage Cache 104 is used:

♦ Level One (L1) of said Cache 104 saves pre-cached information in form of the following tupel (serviceProviderId, serviceUserId, FromSIPAddress, ToSIPAddress, E2ENPSessionID), where the FromSIPAddress and ToSIPAddress allow detecting dual seizures; the primary key is the serviceProviderId.

25 ♦ Level Two (L2) of said Cache 104 saves pre-cached information in form of the following tupel (serviceUserId, E2ENPSessionID), where the primary key is the serviceProviderId.

30 The two levels of the Cache 104 are implemented as two distinct singleton objects as described in [Gam], which are initiated and managed by the E2ENP UA 128. The Cache objects run within the instance of the E2ENP UA 128. Fig. 9 depicts a UML

Class Diagram of the org::mind::e2enp::Cache sub-package that contains the Cache implementation.

The singletons LevelOneCache 902 and LevelTwoCache 904 respectively represent the Cache level L1 and the Cache level L2 and provide APIs for creating, searching, and releasing Cache entries (LevelOneEntry 906 and LevelTwoEntry 908, respectively).

10 The methods provided by the API of the first and second Cache level (L1 and L2) are depicted in Tab. 5 and Tab. 6, respectively.

15 In the following subsection, an XML-based serialized transport representation shall be defined by means of an XML schema definition. Rather than defining such a format from scratch, the transport representation is based upon and using the baseline SDPng format definition. Aside from providing compatibility, it is also possible to use and benefit from a 20 third-party library and profile definitions designed for SDPng as described in [Kut2]. Examples include but are not limited to the Real-Time Protocol (RTP) as well as audio and video codec profiles.

25 In order to illustrate how the Factory API 101e and the Parser API 101d share common transport representation syntax, the following subsection provides one possible definition for such a format. The XML description of E2ENP extends and uses the baseline SDPng XML description format.

30

The XML representation for E2ENP (in the following, E2ENP will be used synonymously for this representation) has chosen to be an extension to the SDPng baseline XML definition. The main reason for this decision is that SDPng is expected to be

standardized by the IETF, which results in some major benefits:

- ◆ E2ENP as an extension to SDPng will be able to benefit from external contributions to SDPng-like profile and library definitions. There are already examples pointed out in the SDPng draft [Kut2], namely the audio and RTP profile. These extensions are already incorporated into the specification of the E2ENP XML.

10

- ◆ Applications 130 that implement the future SDPng standard can more easily be enhanced to also support E2ENP, ideally by simply linking the E2ENP module to the application 130. Likewise, in heterogeneous environments, applications 130 which do not support E2ENP are still able to work with the SDPng-specific parts and do not break interoperability completely.

15

The E2ENP XML representation is formally specified by an XML Schema definition as defined by the W3C. The schema definition can roughly be separated into three parts, one containing type definitions such as enumerations for attribute values or the definition of common content models as complex types. The main part defines the top-level E2ENP sections (i.e. elements), which are also plugged into SDPng using the substitution group mechanism. Finally, the third part defines E2ENP-specific elements with no relation to SDPng, which are used to completely define the content model of the top-level elements.

20

As can be seen in Fig. 10, descType 1001 - originally defined in SDPng - is redefined in such a way that the E2ENP-specific header section „purpose“ 1002 appears as a legal child of the „desc root“ element. E2ENP defines a „qosdef“ section which 35 can be used as a substitute for the „sdpng:d“ element, the

only valid child for the „sdpng:def“ 1003 section, similarly, „qoscfg“ may be used instead of „sdpng:c“, in turn the only valid child for the „sdpng:cfg“ 1004 section. The E2ENP-specific element types will be described in the following.

5

The extensions and plug-ins to SDPng are illustrated in Fig. 11, which shows how elements in substitution groups can be used to replace the head element of this group.

10 Here, „sdpng:d“ 1107 is the head element which may be substituted by any of the members of the substitution group, that is, „audio:codec“ 1108, „e2enp:qosdef“ 1109, „rtp:pt“ 1110, „rtp:transport“ 1111 (which in turn again is the head element of a nested substitution group) and „video:codec“ 1113. Thus,

15 by using the substitution group mechanism combined with the XML namespaces concept, it is possible to define modular extensions to any XML schema definition. Additionally, the substitution group member „rtp:udp“ 1112 can further be enhanced in order to support the option sub-group described in [Bos2],

20 thereby incorporating an option similar to that suggested by [Bos2] (not shown in Fig. 11).

25 The aforementioned „qoscfg“ element 1400 is defined in the substitution group headed by „sdpng:c“, which in turn is a valid child of the „sdpng:cfg“ section.

Using the „purpose“ 1201 element depicted in Fig. 12, it is possible to convey additional information about the negotiation process in the external description. The general structure of how the purpose element is defined is shown in Fig. 30 12.

Thereby, the header defines to which session the current message belongs to. It can also establish references to previous

sessions by means of the „use“ 1202 element, making it possible to reference definitions from these sessions.

Depending on what the E2ENP message is intended for, either  
5 the „description“ 1203 or the „mediation“ 1204 element should  
be present. For example, with the aid of the „description“  
1203 element it is possible to state the kind of a message,  
i.e. Request (Req) or Response (Rsp) and what negotiation  
mode should be used (push, pull, push-pull).

10 Fig. 13 illustrates the general structure of the „qosdef“  
element 1300. However, there is a constraint on the usage of  
the legal child elements, depending on the value of the  
„name“ attribute of the qosdef element 1300. There are two  
15 possible values, „capabilities“ and „contracts“. Depending on  
that, the „qosdef“ section 1300 either specifies capabilities  
of a peer or, in the case of contracts, defines validated  
contracts that have been validated by entities using the  
E2ENP UA 128.

20 – Specifying Capabilities: In case the „name“ attribute's  
value of the „qosdef“ element 1300 is „capabilities“, the  
valid child elements are „audio:codec“ 1301, „video:codec“  
1302, and „rtp:pt“ 1303. The respective codec elements rep-  
25 resent system capabilities of the peer in terms of audio  
and video codecs and their configuration. By using the  
„rtp:pt“ 1303 element, it is possible to specify a desired  
RTP payload format for the given capabilities.

30 – Specifying Contracts: In this case, the attribute value of  
the „name“ element of the qosdef element 1300 has to be  
„contracts“. Then, the valid child elements are „policy“  
1304, „audio“ 1305, and „video“ 1306. The „policy“ element  
has to appear exactly once. It specifies the usage for ne-  
35 gotiating the resource management policies to enforce. It

is possible to use a policy to optimize one or several aspects of system behavior, such as optimization of memory usage, CPU load, network traffic or power consumption. In order to achieve more flexibility, these atomic aspects can be combined by using predicates. The elements following the „policy“ 1304 element can be one or many „audio“ 1305, „video“ 1306, „data“ 1308, and „control“ 1307 elements. These elements describe the QoS Contracts for the respective data streams. By using the „rtp:map“ 1309 element it is possible to define a mapping between an application level QoS Contract and a specific RTP payload format.

Fig. 14 illustrates the general structure of the „qoscfg“ element 1400. This section allows defining adaptation paths (AP) as well as QoS correlation and time synchronization constraints at various levels of abstraction, starting from stream-level QoS Contracts. Each level of abstraction is identified by the attribute „name“ of this section. Possible values include „stream“, „stream-group“, „session“, and „application“. Thus, the definition of adaptation paths is possible at different levels, including stream-level adaptation paths as well as higher-level APs. The „context“ 1401 elements define possible associations of the given streams, thus allowing to define time-synchronization and/or QoS correlation constraints. As such, the „context“ 1401 elements basically describe high-level QoS Contracts.

In the following subsection, a brief overview of the Parser API 101d shall be given.

The Parser API 101d provides primitives, which can be used to parse a description in some transport representation and generate an object representation according to IF2. In order to be able to loosely couple the E2ENP UA 128 and the Parser 112, this object representation is abstracted in such a way

that it is expressed as a `java.lang.Object`. As already pointed out, the only requirement for the transport representation is that it implements the `java.io.Serializable` interface. This can be interpreted in the sense that the transport 5 representation is un-typed, thereby providing much flexibility regarding what this representation can actually look like.

The transport representation may contain references to external, previously defined entities. These references are left untouched and unresolved by the Parser 112. This issue is 10 handled later in the E2ENP UA 128 or higher layer entities.

The Parser API 101d provides one primitive for each message type to be passed. Like this, the message types are represented implicitly and do not need to be considered in the object model. For each message type, the Parser 112 offers one primitive to parse the offer and the answer of the respective message type. Therefore, the general pattern depicted in Fig. 15 20 15 for the Parser API 101d primitives is

```
Object parseXXX(Serializable input).
```

Thereby, „XXX“ denotes a placeholder for the names of primitives like `NegOffer` or `PreNegAnswer`, etc. (see the API definition below for more details). 25

In addition to these general parsing methods, the Parser API 101d provides some specialized primitives intended to navigate through the content wrapped in the serialized representation. These primitives can be used by clients to access specific information or data on an abstract level, independent from the actual representation. Thus, if the representation changes due to future protocol enhancements, the clients 30 35 which use older versions of the representation are not af-

fected by the new changes since the information access methods do not change.

As the actual carrier protocol should be exchangeable, there  
5 is a need for providing functionality to exchange the actual  
Parser 112 implementation as well. As described above, this  
mechanism has to be the same for the Parser 112 implementa-  
tion and for the carrier protocol.

10 Therefore, a ParserFactory API is defined which provides  
primitives to create an actual Parser 112 implementation in-  
stance (factory 118 depicted in Fig. 1 shows an implemen-  
tation of such a ParserFactory). Additionally, it is possible  
to register new actual Parser 112 implementations for any  
15 given carrier protocol.

The communication between the E2ENP UA 110 and the Parser 112  
could be asynchronous, e.g. implemented by an active request :  
interface and a passive confirmation interface. However, this  
20 option is not appropriate for the underlying design, because  
some gain in performance and flexibility is opposed by much  
more complicated state models for the FSMs 106 of the E2ENP.  
One consequence of this decision is that the E2ENP UA 128 and  
the Parser 112 (and also the Factory 114) communicate syn-  
25 chronously and thus have to share a common address space,  
which in most circumstances is a reasonable assumption.

Another consequence of this decision is that the Parser 112  
(and also the Factory 114) have to be designed as thread-  
30 safe, re-entrant libraries.

The object representation, generated during the parsing and  
passed as a result back to the calling E2ENP UA 128, is de-  
scribed very abstractly by just using the java.lang.Object  
35 class, in order to provide loose coupling of the Parser 112

and the E2ENP UA 128. The actual classes implementing the Parser 112 for a given transport representation and a given object model, used in the E2ENP UA 128, have to be fitted together by configuration.

5

The actual Parser and Factory implementations 112 and 114, respectively, have to agree on a common format for the transport representation. The definition of these formats is implementation-specific and corresponds to the actual Parser 112 and Factory 114 components, even though the usage of existing standards is encouraged and envisioned. Therefore, a format has been defined, which is compatible to the baseline SDPng definition.

15 The registration functionality offered by the ParserFactory API can be handled in different ways. However, this is up to the actual ParserFactory 112 implementation. Some options what can be done with registration are:

20 ◆ Registrations may be stored permanently to survive multiple application life cycles.  
◆ Registrations may be made public for the use of other applications 130, which also implies permanent storage.  
◆ Registrations may only be stored in a transient way for the  
25 current application life cycle.

Applications 130 may check if they have instantiated a matching pair of Parser 112 and Factory 114 by calling the respective getType() methods of these instances and comparing the 30 returned values for equality. The usage of a non-matching pair of Parser 112 and Factory 114 by an application 130 may result in unpredictable results and undefined or even erroneous behavior.

In the following, the dependencies shall briefly be summarized.

◆ IF2 Object Model: This is only an implicit dependency that  
5 actual implementations of the Parser 112 have to fulfill.

In general, this dependency is removed by the fact that  
the Parser 112 simply creates java.lang.Object instances.

In a specifically configured system however, which will  
actually be created, instances of the specific object  
model classes are used in the E2ENP UA 128.

◆ Transport Representation: Similar to the dependency above,  
an actual Parser 112 of course needs a well-defined struc-  
ture of what it has to create. Thus, it depends on the re-  
15 spective definition of the transport representation. But  
in general, by abstractly describing the transport repre-  
sentation using java.io.Serializable, this dependency is  
removed from the design.

20 The primitives to be implemented by the Parser API 101d can  
be taken from Tab. 7. By contrast, the primitives to be im-  
plemented for generating a specific Parser 112 are listed in  
Tab. 8.

25 The ParserFactory API is realized by the class E2ENPContent-  
HandlerFactory in the package org.mind.e2enp.content, also  
referred to as parser-factory implementation. It is designed  
as a singleton instance, so that all its clients can use reg-  
istrations made earlier. Currently, the singleton instance  
30 created is a default implementation, but for future releases  
it is planned to provide a configurable approach (based on  
specific system properties) to determine the class to be used  
for the singleton instance. Methods provided by the E2ENP-  
ContentHandlerFactory class are depicted in Tab. 11.

In the following subsection, the implementation of the Parser 112 needed for an XML-based transport representation shall briefly be described. Thereby, a possible Parser 112 implementation for the XML-based transport representation is proposed. Fig. 17 describes the general structure of a Document Object Model (DOM) tree, modeled as a class DocumentTree 1702, which aggregates one DocumentNode 1704 and can be interpreted as the document root element. Each DocumentNode 1704 in turn aggregates between 0 and N children, thus recursively describing the tree structure.

The Parser 112 for this representation can be implemented by using the „visitor design pattern“ as described in [Gam]. The structural overview is given in Fig. 18. Thereby, a generic Parser 112 visitor class pVisitor 1808 is defined to use, or to be precise visit, 1 to N DOMNodes 1806. In order to handle derived DOMNodes for each specially derived DOMNode subclass, a specialized visitor is defined. (Even if the „visitor pattern“ is intended for handling only subclasses, this concept can easily be extended in such a way that different element nodes are interpreted as categories of their own.)

In the following, the Application Programming Interface (API) applied to the Factory 114 shall briefly be described.

The Factory API 101e provides primitives which can be used to create a transport representation for a given object representation according to IF2. The loose coupling of the E2ENP UA 128 and the Factory 114 is realized via the abstraction of the object representation (passing through the Factory interface 101e) as a java.lang.Object. As already pointed out, the only requirement for the transport representation is that it implements the java.io.Serializable interface. Therefore, the primitives defined by this API all return „java.io.Serializable“ objects.

The object description may contain references to external, previously defined entities. These references are not handled or dereferenced by the Factory methods, they are just used.

5

Like the Parser 112, the Factory API 101e provides one primitive for each message type to be passed. Like this, the message types are represented implicitly and do not need to be considered in the object model. The general pattern for the

10 Factory API primitives as depicted in Fig. 19 is:

```
Serializable createXXX(Object input)
```

Thereby, „XXX“ denotes a placeholder for the names of primitives like NegOffer or PreNegAnswer, etc. (see the API definition below for more details).

Since the actual carrier protocol should be exchangeable, there is a need to provide a functionality to make the actual  
20 Factory 114 implementation exchangeable as well. As described above, this mechanism has to be the same for the Factory 114 implementation and for the carrier protocol. Therefore, a FactoryFactory API is defined which provides primitives to create an actual Factory 114 implementation instance.  
25 (factory 120 in Fig. 1 shows an implementation of such a FactoryFactory.) Additionally, it is possible to register new actual Factory 114 implementations for any given carrier protocol in order to illustrate how the Factory API 101e and the Parser API 101d share a common transport representation syntax.

30 Communication between the E2ENP UA 128 and the Factory 114 could be asynchronous, e.g. implemented by an active request interface and a passive confirmation interface. This option  
35 is not used in the scope of the underlying invention because

some gain in performance and flexibility is opposed by much more complicated state models for the FSMs 106 of the E2ENP. One consequence of this decision is that the E2ENP UA 128 and the Factory 114 (also the Parser 112) communicate synchronously and thus have to share a common address space, which in most circumstances is a reasonable assumption.

Another consequence of this decision is that the Factory 114 (and also the Parser 112) have to be designed as thread-safe, re-entrant libraries.

In order to provide loose coupling of the Factory 114 and the E2ENP UA 128, the transport representation, generated by the Factory 114 and passed back to the calling E2ENP UA 128 as a result, is abstractly described by using the java.io.Serializable class. Similarly, the java.lang.Object class abstractly describes the inputs of the Factory 114 primitives. Thus, the actual classes implementing a Factory 114 for a given transport representation and a given object model used in the E2ENP UA 128 have to be fitted together by configuration.

The actual Parser 112 and Factory 114 implementations have to agree on a common format for the transport representation. The definition of these formats is implementation-specific and corresponds to the actual Parser 112 and Factory 114 components, even though the usage of existing standards is encouraged and envisioned. Therefore, a format is proposed which is compatible to the baseline SDPng definition.

30

The registration functionality offered by the FactoryFactory 120 can be handled in different ways. Some options for registrations include but are not limited to:

- ◆ Registrations may be stored permanently to survive multiple application life cycles.
- ◆ Registrations may be made public for the use of other applications 130, which also implies permanent storage.

5 ◆ Registrations may only be stored in a transient way for the current application life cycle.

Applications 130 may check if they have instantiated a matching pair of Parser 112 and Factory 114 by calling the respective getType() methods of these instances and comparing the returned values for equality. The usage of a non-matching pair of Parser 112 and Factory 114 by an application 130 may result in unpredictable results and undefined or even erroneous behavior.

15

In the following, the dependencies shall briefly be summarized:

- ◆ IF2 Object Model: This is only an implicit dependency that actual implementations of the Factory 114 have to fulfill. In general, this dependency is removed by the fact, that the Factory 114 simply expects java.lang.Object instances as input data. In a specifically configured system, however, which actually will be passed in to the creation process, instances of the specific object model classes are used in the E2ENP UA 128.
- ◆ Transport Representation: Similar to the dependency above, an actual Factory 114 of course needs a well-defined structure of what it has to create. Thus, it depends on the respective definition of the transport representation. But in general, by abstractly describing the transport representation using java.io.Serializable, this dependency is removed from the design.

The primitives specified by the Factory API 101e can be taken from Tab. 9. Additionally, the primitives to be implemented for generating a specific Factory 114 are listed in Tab. 10.

5

The FactoryFactory API is realized by the class E2ENPContentHandlerFactory in the package org.mind.e2enp.content, also referred to as factory-factory implementation. It is designed as a singleton instance, so that all its clients can use registrations made earlier. Currently, the singleton instance created is a default implementation, but for future releases it is planned to provide a configurable approach (e.g. based on specific system properties) to determine the class to be used for the singleton instance. Methods provided by the E2ENPContentHandlerFactory class can be taken from Tab. 11.

10

In the following subsection, a possible Factory 114 implementation for the XML-based transport representation shall be presented.

20

Like the Parser 112, the Factory 114 implementation uses the "visitor design pattern" described in [Gam]. The structure is illustrated by Fig. 20.

25

Again, a generic Factory 114 visitor class fVisitor 2002 is defined which visits 1 to N instances of ObjectGraphNode 2004 classes. For each concrete subclass of ObjectGraphNode, a corresponding concrete visitor class is defined.

30

In the following subsection, a brief overview of the SIP User Agent Generic API 101c (IF3) shall be given.

35

Since the SIP User Agent Generic API 101c exposes the functionality of a generic SIP UA 110 to the E2ENP UA 128, the

main object of this API is to mask the actual implementation of the SIP UA 110 from the E2ENP UA FSM 106. The SIP User Agent Generic API 101c realizes the specification of such generic API by defining a set of interfaces and complex data types exchanged through said API. Specific SIP UA 110 implementations will be able to expose their native (eventually standard) APIs by means of specific Adapters, which shall map the SIP API native interfaces into the SIP UA Generic API 101c, and vice versa.

10

The SIP UA Generic API 101c allows multiple users to simultaneously access a given SIP UA 110 implementation (via multimedia applications 130 or middleware 130) for simultaneously establishing multiple SIP sessions.

15

The SIP UA Generic API 101c also supports SIP Registrar implementations with the SIP signaling required by these entities. To this extent, it should be noted that the SIP UA Generic API 101c is not designed to allow the simultaneous support of users' applications 130 and SIP Registrars on the same memory address space. With this design choice, SIP Registrar implementations are thus forced to run as standalone server side entities, distinct from pure application 130 or middleware 130 entities (in the present context, the E2ENP UA FSM 106). Applications 130 and/or middleware 130 (in the present context, the E2ENP UA FSM 106), and SIP Registrar implementations are examples of Service Users. A SIP UA 110 implementation exposing the SIP UA Generic API 101c represents the Service Provider.

20

The SIP UA Generic API 101c has originally been designed to expose an API from a specific open source SIP UA 110 implementation, the Mitre's JSIP v.0.8 library (<http://jsip.sourceforge.net/>), wherein API support is quite

poor. The SIP UA Generic API 101c has been designed based on the following design principles:

1. The definition of a set of primitives identifying various SIP procedures abstracted at a relatively high-level, thereby simplifying the E2ENP UA FSM 106 design. Of course, the granularity of the procedures exposed by the current version of the SIP UA Generic API 101c reflects to a certain extent the capability of the JSIP implementation. Other SIP UA 110 implementations might offer more or less capabilities, which would require respectively more or less logic embedded in the corresponding aforementioned Adapters.
2. The current version of the SIP UA Generic API 101c focuses only on the minimal set of features required for prototyping and testing the E2ENP UA 128. Therefore, the current version of SIP UA Generic API 101c is on purpose neither a complete SIP API specification nor meant to be standardized whatsoever.
3. Primitives are classified into Request (Req), Indication (Ind), Response (Rsp), and Confirmation (Cfm) primitives according to the ISO/OSI model. Request (Req) and Response (Rsp) primitives are invoked by the client Service User of the IF3 101c, or by a SIP Registrar implementation in order to respectively initiate or respond to a particular message exchange. (In some cases the message exchange may involve a set of different messages being exchanged between peers, thanks to the abstraction offered by the given SIP UA 110 implementation, whereas in other cases the message exchange is limited to the given message and its acknowledgement.) Indication (Ind) and Confirmation (Cfm) primitives are invoked by the Service Provider 110 to the

5 registered client-side of the Service User of the IF3  
101c in order to respectively indicate the arrival of a  
particular message or initiation of a particular mes-  
sage exchange or to confirm the conclusion of a given  
message exchange. The client-side of the Service User  
of the IF3 101c has to implement these primitives. To a  
Request (Req) primitive generated by the peer initiat-  
ing the given protocol message exchange corresponds an  
Indication (Ind) primitive at the target peer. To a Re-  
10 sponse (Rsp) primitive generated by the target peer,  
responding to the given protocol message exchange, cor-  
responds a Confirmation (Cfm) primitive at the initiat-  
ing peer.

15 4. A simplified version of the „observer design pattern“  
as described in [Gam] (implemented in Java with a deri-  
vation of the Java Beans event model) offers the mecha-  
nism allowing the Service User to register (via a  
primitive called „Bind Request“) for Indication (Ind)  
20 and/or Confirmation (Cfm) primitive callbacks.

25 5. In order to allow supporting E2ENP piggybacking over  
SIP, each primitive allows passing a payload, e.g. SDP,  
SDPng, or Multipurpose Internet Mail Extension (MIME)  
content.

30 6. In order to allow dry-run tests of the E2ENP signaling  
logic without the support of a real SIP UA 110 imple-  
mentation and of SDP/SDPng Parsers 112 and/or Factories  
114, the primitives are designed to handle the payloads  
as generic objects (instead of plain objects represent-  
ing ASCII strings) that can be marshalled or demar-  
shalled when using Remote Procedure Call (RPC) mecha-  
nisms, like Java serialization or deserialization in  
35 combination with Java RMI.

7. Since Mitre's JSIP Version 0.8 implementation used for designing this SIP UA Generic API 101c lacks proper support of the SIP Expires header field, which is typically manipulated by Service Users, an abstraction of this SIP header field has been created for handling any type of implementation of this field. This would be not necessary with other better SIP UA 110 implementations, where the designer would have the choice to either change the SIP UA Generic API 101c (and therefore the E2ENP UA FSM 106 implementation) accordingly or provide an Adapter 126c mapping the Expires header field support of the given Service Provider (without changing the E2ENP UA FSM 106 implementation).

15

Further design choices are:

- The E2ENP negotiation model described in [ReqSpec] is a key aspect of the E2ENP and a piece of information intended for being embedded into SDPng payload and exchanged among peers on an end-to-end basis. For the sake of rapidly prototyping the E2ENP UA 128, a helper class named SipNegotiationModel 2216, representing the E2ENP negotiation model information, has provisionally been included into the SIP UA Generic API 101c.
- Instead of uniquely identifying active sessions at application level by using the alphanumeric SIP Call Identifier header field, the SIP UA Generic API 101c uses a numerical integer identifier, named connectionId. This choice was dictated by the following reasons:
  - Service users can more easily handle lists with numerical identifiers rather than with alpha-numerical ones.

35

Should the E2ENP UA 128 use other protocols instead of SIP in future, having generic primitives and generic session identifiers would help partly reusing the existing E2ENP UA FSM 106 design.

5

The SIP UA 110 associates an unused value of the connectionId to the SIP Call Identifier header field of an incoming INVITE, OPTIONS, MESSAGE, or REGISTER method. By applying the same design principle, Service Users requesting the transmission of any of those methods would obtain the connectionId value selected by the Service Provider in the Confirmation (Cfm) primitive associated with the original Request (Req) primitive.

10

15

Unfortunately, this solution is affected by two problems:

20

- Without yet knowing the connectionId value associated with a given Request (Req) primitive, Service Users can not correlate a Confirmation (Cfm) primitive with its corresponding Request (Req) primitive.

25

- Before the arrival of a Confirmation (Cfm) primitive, intermediate messages (eventually indicating failure situations) might generate additional primitives (e.g. a Connection Status Indication), which would lead to the same correlation problem described in the previous point.

30

To solve these problems, the Service User must be in charge of selecting an unused connectionId value and force the SIP UA 110 associating such value with a proper SIP Call Identifier header field value.

35

The set of connectionId values allocated by the Service Provider in case of incoming INVITE, OPTIONS, MESSAGE, or REGISTER methods and by the Service User in case of outgoing INVITE, OPTIONS, MESSAGE, or REGISTER methods should be distinct, so that the processing of incoming and outgoing INVITE, OPTIONS, MESSAGE, or REGISTER methods can simultaneously take place without interference.

However, for the sake of simplicity, the SIP UA Generic API 101c has been designed with a centralized management of connectionId values allocation, embedded into the Service Provider itself. Service users are yet in charge of allocating connectionId values for outgoing INVITE, OPTIONS, MESSAGE, or REGISTER methods, but the actual allocation is delegated to the Service Provider by invoking the getConnectionId() method exposed by Service Provider. This method can in no way be considered a primitive itself, since it directly returns a value (the selected connectionId value): For instance, this would prevent implementing a loosely coupled interface, where primitives are exchanged between Service User and Service Provider via a (not RPC-like) message-based protocol. In any case, Service Users and Service Providers can be designed without the centralized connectionId value allocation management, and yet being compliant with the SIP UA Generic API 101c, by simply not using respectively implementing the getConnectionId() method. As a special case, a connectionId can be reused in case of SIP re-INVITES: in this case, a proper Boolean parameter indicating whether this is a re-INVITE, shall be passed to the connectionReq primitive (see below).

- Some primitives of the SIP UA Generic API 101c are SIP-specific:

- `provisionalAck(Req|Ind|Rsp|Cfm)`, which is used to implement the PRACK method,
- `register(Req|Ind|Rsp|Cfm)`, which is used to implement the REGISTER method,
- 5     • `options(Req|Ind|Rsp|Cfm)`, which is used to implement the OPTIONS method, and
- `message(Req|Ind|Rsp|Cfm)`, which is used to implement the MESSAGE method.

10       These primitives will eventually be mapped to more generic ones in a future E2ENP UA 128 design review in order to transform the SIP UA Generic API 101c into a E2ENP Session Level Protocol API, which will be totally independent of the session-layer protocol actually used.

15       • The current version of the SIP UA Generic API 101c design is only partially compliant with the latest working (as of this writing) version of the SIP specification described in [Ros1]. For instance, REFER, DO, NOTIFY, SUBSCRIBE methods and many other features have been omitted since they are not essential for testing the E2ENP concept. The BYE and CANCEL methods are supported by the same Disconnection primitive.

25       Thereby, said E2ENP UA API 101b depends on the applied E2ENP UA FSM 106, concerning the implementation of the Indication (Ind) and Confirmation (Cfm) primitives and the Management component 102, which is used for configuring, controlling, and resetting the underlying SIP UA 110.

30       The SIP UA Generic API 101c has been designed using the object-oriented paradigm, and is thus hereby described by using the Unified Modeling Language (UML). It is composed of a basic package, the so-called `org::mind::sip::sipApi` as depicted  
35       in Fig. 21. To this basic package belong the classes `SipEnd-`

User 2214 and SipExpiresHandling 2112. The SipEndUser class 2214 represents a user accessing the SIP services offered by the SIP UA Generic API 101c. The SipExpiresHandling class generalizes the functionality for manipulating the SIP Expires header field. The SipListener 2108 generalizes all the interfaces that Service Users shall implement for intercepting Indication (Ind) and/or Confirmation (Cfm) primitives generated by the SIP UA 110. The SipProvider 2110 generalizes all the interfaces that the Service Provider shall support 10 for implementing Request (Req) and Response (Rsp) primitives.

Four sub-packages complete the SIP UA Generic API 101c specification: one, the org::mind::sip::sipApi::time, deals with the SIP Expires header field abstraction, whereas the other 15 three characterize different roles of the Service User:

- org::mind::sip::sipApi::userAgent represents the application 130 / middleware 130 implementation (in the present context, the E2ENP UA FSM 106);
- org::mind::sip::sipApi::management represents the management entity 102;
- org::mind::sip::sipApi::registrar represents the SIP Registrar implementation 2306.

The org::mind::sip::sipApi::userAgent sub-package (cf. Fig. 22) contains interfaces and classes specifying the part of the SIP UA Generic API 101c that the Service User of the IF3 101c (in the present context, the E2ENP UA FSM 106) use for accessing the Service Provider. It consists of three major parts:

1. Interfaces that Service Users of the IF3 101c shall implement for intercepting the Indication (Ind) and Confirmation (Cfm) primitives generated by the Service Provider 110:

- SipUserAgentListener 2202: common to both client and server side; a specialization of the org::mind::sip::sipApi::SipListener interface 2108.
- 5 - SipUserAgentClientListener 2206: specific for the client side; a specialization of the SipUserAgent-Listener interface 2202.
- SipUserAgentServerListener 2204: specific for the server side; a specialization of the SipUserAgent-Listener interface 2202.

10

2. Classes bundling parameters for specific primitives (currently only the SipConnectionReq class 2208 for the Connection Request (Req) primitive, the SipRegistration-Req class 2218 for the Registration Request (Req) primitive, and the SipRegistrationCfm class 2220 for the Registration Confirmation (Cfm) primitive). Instances of these classes are passed as parameters of the given primitives to the modules implementing the primitives. This solution allows extending and/or changing API details without disrupting the primitive signatures.

15

3. Interfaces and classes specifying the API that the Service Provider shall support for being compatible with the SIP UA Generic API 101c. The SipUserAgentType interface 2210 specifies the set of Request (Req) and Response (Rsp) primitives that the underlying SIP UA 110 implementation shall support for being compatible with the SIP UA Generic API 101c. The SipUserAgentType interface 2210 is a specialization of the

20

org::mind::sip::sipApi::SipProvider

25

interface. The SipUserAgent class 2214 wraps any given implementation of the aforementioned SipUserAgentType

30

35

interface 2210, by applying the „abstract factory“ and „singleton“ design patterns as described in [Gam]. The „abstract factory“ is defined by the SipUserAgentFactory interface 2212. The SipUserAgent class 2214 is designed 5 to use a default implementation by using the defImpl class attribute holding the fully qualified class name of the default implementation. Custom implementations of the aforementioned SipUserAgentType interface 2210 can be created by providing specific factory classes imple- 10 menting the aforementioned SipUserAgentFactory interface 2212. The SipUserAgent class 2214 provides a class method called setUserAgentFactory(), which stores a sin- gleton instance of the given custom factory.

15 Finally, this sub-package contains a class named SipNegotiationModel 2216, which represents the E2ENP negotiation model.

The org::mind::sip::sipApi::registrar sub-package (cf. Fig. 23) contains interfaces and classes specifying the part of 20 the SIP UA Generic API 101c that SIP Registrar implementa- tions used for accessing the Service Provider. It consists of three major parts:

25 1. Interfaces that Service Users of the IF3 101c shall im- plement for intercepting the Indication (Ind) and Con- firmation (Cfm) primitives generated by the Service Provider:

30 SipRegistrarListener 2302: common to both client and server side; a specialization of the

org::mind::sip::sipApi::SipListener

interface 2108.

2. Classes bundling parameters for specific primitives (currently only the SipRegistrationInd class 2308 for the Registration Indication (Ind) primitive, and the SipRegistrationRsp class 2310 for the Registration Response (Rsp) primitive). Instances of these classes are passed as parameters of the given primitives to the modules implementing the primitives. This solution allows extending and/or changing API details without disrupting the primitive signatures.

10

3. Interfaces and classes specifying the API that the Service Provider 110 shall support for being compatible with the SIP UA Generic API 101c. The SipRegistrarType interface 2304 specifies the set of Request (Req) and Response (Rsp) primitives that the underlying SIP UA 110 implementation shall support for being compatible with the SIP UA Generic API 101c. The SipRegistrarType interface 2304 is a specialization of the

15

org::mind::sip::sipApi::SipProvider

20

interface 2110. The SipRegistrar class 2306 wraps any given implementation of the aforementioned SipRegistrarType interface 2304 by applying the „abstract factory“ and „singleton“ design patterns as described in [Gam]. The „abstract factory“ is defined by the SipRegistrarFactory interface 2312. The SipRegistrar class 2306 is designed to use a default implementation by using the defImpl class attribute holding the fully qualified class name of the default implementation. Custom implementations of the aforementioned SipRegistrarType interface 2304 can be created by providing specific factory classes implementing the aforementioned SipRegistrarFactory interface 2312. The SipRegistrar class 2306 pro-

25

30

vides a class method called `setRegistrarFactory()`, which stores a singleton instance of the given custom factory.

The `org::mind::sip::sipApi::management` sub-package (cf. Fig.

5 24) contains interfaces and classes specifying the part of  
the SIP UA Generic API 101c applied to the management entity  
102 for configuring and controlling the Service Provider. It  
consists of three major parts:

10 1. Interfaces that Service Users shall implement for intercepting the Indication (Ind) and Confirmation (Cfm)  
primitives generated by the Service Provider:

SipManagementListener.

15 2. Classes bundling parameters for specific primitives  
(currently only the `SipConfigurationReq` class 2406 for  
the Configuration Request (Req) primitive). Instances of  
these classes are passed as parameters of the given  
20 primitives to the modules implementing the primitives.  
This solution allows extending and/or changing API de-  
tails without disrupting the primitive signatures.

25 3. An interface specifying the API that the Service Pro-  
vider shall support for being compatible with the SIP UA  
Generic API 101c. The `SipManager` interface 2404 speci-  
fies the set of Request (Req) and Response (Rsp) primi-  
tives that the underlying SIP UA implementation shall  
support for being compatible with the SIP UA Generic API  
30 101c.

The org::mind::sip::sipApi::time sub-package (cf. Fig.. 25) contains interfaces and classes specifying the part of the SIP UA Generic API 101c that allow Service Users manipulating the SIP Expires header field. It consists of two major parts:

5

1. Interfaces and classes specifying the API that the Service Provider shall support for being compatible with the SIP UA Generic API 101c. The SipExpiresType interface 2502 specifies the set of SIP Expires header field manipulation operations that the underlying SIP UA 110 implementation shall support for being compatible with the SIP UA Generic API 101c. The SipExpires class 2506 wraps any given implementation of the aforementioned SipExpiresType interface 2502 by applying the „abstract factory“ and „singleton“ design patterns as described in [Gam]. The „abstract factory“ is defined by the SipExpiresFactory interface 2508. The SipExpires class 2506 is designed to use a default implementation by using the defImpl class attribute holding the fully qualified class name of the default implementation. Custom implementations of the aforementioned SipExpiresType interface 2502 can be created by providing specific factory classes implementing the aforementioned SipExpiresFactory interface 2508. The SipExpires class 2506 provides a class method called setSipExpiresFactory(), which stores a singleton instance of the given custom factory.
2. The SipParseException class 2504 represents the exception thrown by the Service Provider whenever the parsing of the SIP Expires header field fails, due to malformed syntax.

10

15

20

25

30

In the following subsection, the procedures enabled by the SIP UA Generic API 101c shall be described by using a set of UML Message Sequence Charts (MSCs).

5 Fig. 26 shows how the Service Provider 110 is configured, and how a Service User of the IF3 101c creates and binds as Service User its client and server sub-units with the Service Provider 110. Fig. 27 shows how a SIP session can be established through the SIP UA Generic API 101c according to the  
10 procedures described in [Ros1].

This procedure starts with a connectionReq primitive invoked by a peer acting as calling party (or initiator), triggers a message exchange, which terminates only when the peer acting  
15 as called party (or responder) after having initially received a connectionInd primitive notifying the initiator's invitation to establish a SIP session, eventually accepts such invitation (after a number of negotiation and signaling steps), and replies with a connectionRsp primitive.

20 The responder generates SIP-provisional responses through the connectionStatusReq primitive invocations, which map at initiator's side to connectionStatusInd primitive invocations. The initiator acknowledges provisional responses with the  
25 PRACK method, by using the ProvisionalAckReq/Cfm primitive, which corresponds at responder's side to ProvisionalAck-Ind/Rsp primitive, respectively.

This procedure ends with the initiator receiving a connec-  
30 tionCfm primitive and the responder receiving a connection- StatusInd primitive. The latter is due to the SIP three-way acknowledgment scheme described in [Ros1].

Two procedures in this MSC are assumed to be executed auto-  
35 matically by the Service Provider: The generation of the „100

"Trying" provisional response at responder's side, and the generation of the final ACK signal at initiator's side. The latter can, however, be forced by the Service User itself via the connectionStatusRsp primitive, but this means that the  
5 Service User must accordingly be configured not to generate the ACK signal automatically.

Final SIP responses indicating other reasons than success are generated either automatically by the Service Provider or explicitly by the responder's Service User via the connectionStatusReq primitive. These SIP responses are notified to the initiator's Service User via the connectionStatusInd primitive.  
10

15 Fig. 28 shows how the OPTIONS method is handled via the optionsReq|Ind|Rsp|Cfm primitives. Fig. 29 shows how a SIP session can be released via the disconnectionReq|Ind|Rsp|Cfm primitives. The MSC shows the case of BYE, but the same procedure invoked during a connection establishment generates a  
20 CANCEL.

In the following subsection, the Finite State Machine 106 (FSM) applied to the E2ENP UA 128 shall briefly be described.

25 The core of the E2ENP UA 128 is composed of a Finite State Machine 106 (FSM), which coordinates the overall functionality and implements a part of the aforementioned interfaces. More specifically, the FSM 106 implements the Service User part of the IF3, IF4 and IF5, and the Service Provider part  
30 of the IF1 and IF2. Furthermore, the E2ENP UA FSM 106 makes use of the Cache 104 described above.

The FSM 106 is designed hierarchically as a StateChart. In the Root State (cf. Fig. 30), the FSM 106 handles session-  
35 terminating events, either associated with local or remote

users' decisions. Furthermore, the Root State handles pre-negotiation, lease renewal, negotiation as Offerer/Answerer, re-negotiation, and session release processes. It should be noted that the term „negotiation“ process is used for indicating a multimedia session establishment, intertwined with resource reservation coordination and QoS negotiation, as described in [ReqSpec].

The role of the Offerer is taken by the peer who is initiating a session establishment with QoS negotiation and resource reservation, but also by the peer who decides to initiate a re-negotiation. The role of the Answerer is always taken by the peer who is responding to an invitation to carry out a session establishment process with QoS negotiation and resource reservation, but also by the peer who decides to respond to an invitation to carry out a re-negotiation. This means that a peer initially acting the Offerer role can later take either the Offerer or the Answerer role during a re-negotiation. And vice versa, a peer initially acting the Answerer role can later take either the Offerer or the Answerer role during a re-negotiation.

For the sake of simplicity, Fig. 30 does not detail the handling of primitives associated with IF1 (Management API); neither the internal interfaces with the FSM factory 106 nor the Cache 104 are taken into account.

The following subsections provide a detailed description of the mutex-related procedures shown in Fig. 30.

Lower-level details of the negotiation process as Offerer/Answerer are encapsulated in nested FSMs 106, respectively the NegOfferer sub-state (cf. Fig. 31) and the NegAnswerer sub-state (cf. Fig. 32).

The NegOfferer sub-state contains a nested FSM 106 capturing the logic handling the negotiation process, as seen from the Offerer perspective. The NegAnswerer sub-state contains a nested FSM 106 capturing the logic handling the negotiation process, as seen from the Answerer perspective.

In order to guarantee consistency and avoid deadlocks during the combined reservation of local, peer, and network resources as described in [ReqSpec], the FSMs 106 nested in the NegOfferer sub-state and in the NegAnswerer sub-state use the system-wide „\_ResvMtx“ FSM 106 for accessing the reservation phase on a mutually exclusive basis, with respect to other instances of the E2ENP UA FSM 106. The „\_ResvMtx“ FSM 106 interacts with various instances of the E2ENP UA FSMs 106 via atomic local methods invocation (e.g. via any specific system call provided by the underlying Operating System).

The following subsections provide a detailed description of the mutex-related procedures shown in Fig. 31.

Lower-level details of the re-negotiation process as Offerer/Answerer are encapsulated in nested FSMs 106, respectively the ReNegOfferer sub-state (cf. Fig. 33) and the ReNegAnswerer sub-state (cf. Fig. 34).

- The ReNegOfferer sub-state contains a nested FSM 106 capturing the logic handling the re-negotiation process, as seen from the Offerer perspective.
- The ReNegAnswerer sub-state contains a nested FSM 106 capturing the logic handling the re-negotiation process, as seen from the Answerer perspective.

In order to guarantee consistency and avoid deadlocks during the combined reservation of local, peer, and network re-

sources as described in [ReqSpec], the FSMs 106 nested in the ReNegOfferer sub-state and in the ReNegAnswerer sub-state use the system-wide „\_ResvMtx“ FSM 106 for accessing the reservation phase on a mutually exclusive basis, with respect to 5 other instances of the E2ENP UA FSM 106. The „\_ResvMtx“ FSM 106 interacts with various instances of the E2ENP UA FSMs 106 via atomic local methods invocation (e.g. via any specific system call provided by the underlying Operating System).  
10 The following subsections provide a detailed description of the mutex-related procedures shown in Fig. 33.

This mechanism enforces a transaction-like processing of the resource reservation phase by enforcing various concurrent 15 instances of the E2ENP UA Service Users to mutually exclusively access such a phase, as described in [ReqSpec]. In other words, the resource reservation mutex allows defining the phase between local admission control and final network resource reservation as a critical section. Combined with the 20 contention resolution policy, this mechanism also guarantees that no deadlock situation as described in [ReqSpec] might affect the E2ENP whatsoever.

Concurrent E2ENP UA Service User instances try to seize the 25 mutex via suspensive method calls. The Service User instance owning the mutex releases it via the signal call event.

The „\_ResvMtx“ is associated with a priority queue for allowing multiple requests to seize the mutex in a coordinated 30 manner, based on their priority (cf. Fig. 35).

Any request is served immediately if the mutex is unlocked; otherwise, the request is queued. Requests with high priority are queued in front of requests with low priority. Requests

with the same priority are queued in a „First-In-First-Out“ (FIFO) order.

All requests to acquire the mutex are limited in time. Upon  
5 expiration of such a (configurable) time, the given E2ENP UA Service User instance pending on the mutex is resumed with a return value indicating failure.

This design choice allows detecting (and thus preventing)  
10 starvation of concurrent E2ENP UA Service User instances suspended on the mutex, which occurs when the Service User instance owning the mutex is misbehaving (or maybe blocked). This is of most importance, especially in case the ill-behaved Service User instance are suspended on some other application-specific mutex, semaphore or lock, eventually seized by one of the Service User instances queued on the „\_ResvMtx“: In such an unfortunate situation, a dead lock would in fact inevitably occur.

20 When the timer associated with each blocked request expires, the priority of that request (specified in the parameter list of respectively the bookNegotiationReq or bookRenegotiation-Req primitives, cf. Tab. 1) should be higher than the priority of the one for which the mutex was granted, the mutex  
25 would throw a signal event, congestionInd, to the instance of the FSM 106 UA associated with the Service User instance owning the mutex. This event is caught by the E2ENP UA FSM 106 Root state and mapped to a failureInd primitive sent to the Service User instance owning the mutex.

30

According to application-specific policies, such Service User instance might react on such event by rolling-back the current status of its activities and starting a release operation, which in turn would force the release of the mutex, as  
35 indicated in Fig. 30. Should however such Service User in-

stance act differently (either based on application-specific policies or because being blocked or just misbehaving), the other instances would nevertheless be able to detect the prolonged delay on acquiring the mutex (after a configured number of failed attempts to invoke the get method successfully), thanks to the bookNegotiationCfm (or bookRenegotiationCfm in the re-negotiation case) primitives indicating a failure case (see Figs. 31 and 33, respectively). In such cases, the notified E2ENP UA Service User instances might invoke the mediation of an application-specific arbitration unit. To manage all these exceptional cases/dead locks, the resetReq primitive can be issued at any time to cancel the current given session and release the mutex, if necessary.

15 A key issue typically encountered when using mutex entities is the so-called priority inversion problem, which occurs whenever a task  $T_j$  owning the mutex has a lower priority than other tasks  $T_k$  blocked on the mutex, and more precisely when the  $T_j$  gets pre-empted by a task  $T_l$  with a priority greater  
20 than  $T_j$ , but less than  $T_k$ .

Given the design choice of enforcing loosely coupled interfaces to both the E2ENP UA Service Users and to the SIP UA  
110, the E2ENP UA 128 containing the „\_ResvMtx“ implementation  
25 cannot directly enforce well-known protocols, e.g. the Priority Inheritance Protocol or the Priority Ceiling Protocol. This is after all a consequence of designing the E2ENP UA 128 as a software component.

30 However, if the E2ENP UA FSM 106 is designed to handle a thread-pool for processing concurrently multiple instances of the FSM 106, the priority inversion problem might be tackled at least on this level. In this connection, the enforcement of the Priority Inheritance Protocol allows to achieve a good  
35 performance in handling the aforementioned problem. To this

extent, the priority specified in the parameter list of the bookNegotiationReq primitive (or bookRenegotiationReq in the re-negotiation case, cf. Tab. 1) issued by each E2ENP UA Service User instance is assigned as priority of the thread serving that FSM 106 instance. The „\_ResvMtx“ shall therefore temporarily raise the priority of the thread associated with the FSM 106 instance owning the mutex to the priority of any blocked thread if the priority of the latter is greater than the one of the former.

10

The E2ENP UA 128 thus provides the tools for solving deadlock and priority inversion problems: However, this is up to the applications 130 or QoS Brokers 130 using such tools to guarantee that those problems will never occur.

15

In the following subsection, the resource reservation coordination process shall be described.

The NegAnswerer (cf. Fig. 32) and ReNegAnswerer (cf. Fig. 34) 20 sub-states each contain an instance of the same composite state, the WaitForCoordDone sub-state (cf. Fig. 36), whose nested FSM 106 captures the logic handling the coordination of resource reservation signaling required before the Answerer is actually alerted and the Offerer informed correspondingly thereupon.

The resource reservation coordination process is based on the mechanisms described in [Cam1], with the following differences:

30

1. The preconditions described in [Cam1] can be mapped to the E2ENP „qosdef“ section and its corresponding negotiation process described in [ReqSpec].
2. The E2ENP prescribes the synchronization of the reservation processes between peers (the „confirm“ tag in

35

[Caml]) as always mandatory due to the „Economy Principle“: This means that the end-to-end signaling of the „confirm“ tag (or anything equivalent) is unnecessary and thus shall not be supported by any E2ENP implementation.

5           3. In case of no network resource reservation carried out by the Offerer, the Answerer shall determine this case by examining the preconditions sent by the Offerer, and correspondingly mark the state RemoteTokenAchieved as  
10          active in Fig. 36, so that the Answerer will be able to transparently deal with resource reservation coordination.

15          Impact on SIP implementation (e.g. the introduction of the status response and the new value „preconditions“ for the „Supported“ header field of the OPTIONS method) are left for further study.

20          An important aspect is that the E2ENP described in [ReqSpec] explicitly claims that the Economy Principle is always required, including the concept described in [Caml] of allowing both end-to-end and segmented reservations. In the latter case, enforcing the Economy Principle might not always be desirable. There might in fact be situations where reservation  
25          can be well done locally on beforehand with no cost and/or major impact on session establishment. For instance, some users might be attached to an intranet or wireless LAN, where network resource reservation might be free and starving other users might flexibly be handled. Therefore, the original  
30          E2ENP shall accommodate this case and rely on the applicability of the Economy Principle to allow the aforementioned case.

35          It should be noted that the E2ENP does not imply that a direct interface to reservation entities is available: It is up

to the applications 130 or QoS Broker 130 to access those entities. If the latter wishes to execute a „pre-reservation“ of network resources, it might do that, but the E2ENP signaling would still give the right timing to the applications 130 or QoS Broker 130, thereby informing it when the reservations at all sides have been accomplished according to the Economy Principle, e.g. which signal has reached the level of the desired QoS (even though best effort communications might well have been started on beforehand, if specified by the applications 130 or QoS Broker 130, respectively).

For the sake of simplicity, the model depicted in Figs. 30 to 36 only partially details the handling of unexpected events and error cases. Tab. 18 completes the description of said model by indicating the behavior expected in case any of such events or errors occur.

A set of timers is defined to avoid that the E2ENP UA FSM 106 gets stuck in a state waiting for an event from the Service User and/or the SIP UA 110. The treatment of timers for handling primitives across the E2ENP UA API 101b (T1-T22) does not prescribe the E2ENP UA FSM 106 taking autonomously corrective actions, except for a few cases (missing negotiationReq in Root.NegOfferer.WaitNegReq and renegotiationReq in Root.ReNegOfferer.WaitReNegReq). The E2ENP UA FSM 106 simply notifies the misbehaving Service User about the detected problem via the failureInd primitive upon which the Service User is supposed to clean up and send a releaseReq primitive to the E2ENP UA FSM 106. However, if this counteraction does not succeed, e.g. owing to massive problems at the Service User side, the E2ENP UA FSM 106 would detect this situation upon expiration of yet another timer: in this case, the E2ENP UA FSM 106 finally takes the decision to release the SIP connection and the overall E2ENP session. To manage all these exceptional cases/dead locks, the resetReq primitive can be

issued at any time to cancel the current given session and release the mutex, if necessary.

However, the composite states Root.ReNegOfferer, Root.Neg-

5 Answerer, Root.ReNegOfferer, Root.ReNegAnswerer may detect a later arrival of the missed primitive (eventually triggered by the invocation of the failureInd primitive) by using each an History state (cf. Figs. 31, 32, 33, and 34). With this solution, the late primitive arrival would be caught in the  
10 right original state and processed correctly. This means that whenever the aforementioned timers expire, the aforementioned composite states shall each take care of updating the history to point to the state where the timer expired.

15 Should a SIP primitive fail, specific timers (T101-T106) would detect the failure at the Root.ReNegOfferer, Root.Neg-  
Answerer, Root.ReNegOfferer, Root.ReNegAnswerer composite states: In these cases, the same procedure as described above (based on the failureInd primitive) shall be enforced.

20

For the interface, similar approaches to the potential problem of primitive failure have to be pursued by means of the management entity 102, which is however not detailed in this section.

25

For properly implementing the E2ENP concept, the dual seizure event (peers simultaneously initiating a negotiation session with each other) shall be properly detected and handled in order to allow that consistency is guaranteed and that any  
30 correlation between outgoing and incoming invitations is enforced.

The former issue can be addressed by using a mutex for regulating the access to the Economy Principle phase on a system-wide basis (the "\_ResvMtx" described in the sections above).

For enforcing any correlation, some additional support is required instead. Two possible approaches are envisioned:

a. to use the push-pull model for allowing both Offerer and  
5 Answerer having a chance to send an offer to the peer,  
thus avoiding the possibility of simultaneously generating  
two negotiations in both directions for the same session;

b. to treat the attempt of handling negotiations simultane-  
10 ously in both direction as a double seizure, which re-  
quires a contention resolution policy to be enforced (e.g.  
wait on a random timer on both sides, and then retry).

But this would mean that both peers should use the same  
external representation of the session ID in order to be  
able to detect a double seizure. Otherwise, each of the  
15 two UAs would treat the outgoing and incoming negotiations  
as independent sessions. This would be similar to the case

peer A invited peer B to e.g. a videoconference, while  
peer B would invite peer A to another videoconference,  
whereas the videoconference could actually be the same.

The way E2ENP defines the external representation of the  
session ID described in [ReqSpec] allows the two peers  
creating only distinct IDs. Therefore, the E2ENP-speci-  
fication per se does not allow detecting double seizures  
25 at all.

A reasonable solution to this issue is to enforce the push-  
pull model (first approach) as much as possible and to in-  
clude in the re-negotiation phase the possibility to allow  
30 the Answerer to send an offer to the Offerer at a later time  
(i.e. after the first negotiation has been entirely com-  
pleted).

This means that the E2ENP shall allow the Offerer and/or any  
35 Answerer to negotiate with the other Answerers and/or the Of-

ferer upgraded or new information for establishing QoS-aware multimedia sessions. This means that re-negotiations can be subdivided in two classes:

5     • planned re-negotiations as described originally in [ReqSpec],  
      • unplanned re-negotiations, which are used for negotiating upgraded or new information during the lifetime of a session.

10

An explicit Boolean parameter in the pre-negotiation primitive indicates which of the two classes the given primitive should be used for. Of course, the semantics could be inferred from the information passed, which would be different in the two cases, but the Boolean parameter is a good trade-off between having this distinction, which has explicitly been made for improving performance, and the need to keep the E2ENP UA API 101b and FSM 106 simple by avoiding additional primitives.

15

However, applying the aforementioned solution might not suffice in the general case since a non-E2ENP compliant SIP UA 110 might interfere with E2ENP compliant peers, or one of the participants might try to invite some of the other participants to a parallel multimedia session and use the E2ENP to accomplish this.

SIP loosely tackles the dual seizure problem by suggesting a contention resolution policy (indicated as a „MAY“ requirement in [Ros1]), whereby peers abort the invitation by replying with an Internal Server Error response, carrying a Retry-After SIP header field set to the time after which the remote peer should retry.

20

25

The dual seizure detection is accomplished by matching all messages between each given couple of users, via the To- and From-SIP header fields. This means if a first user is using UA A and sends an INVITE to a second user on UA B, and vice 5 versa, both users can detect the dual seizure by using the couple (To, From) as a unique identifier for correlating the two INVITES.

But this is not exactly the way E2ENP identifies sessions. If 10 users are engaged in two independent videoconferences, the E2ENP can actually distinguish such cases by using the session ID described in [ReqSpec]. In any case, the biggest issue is what would the SIP UA 110 do in such a case, given that the aforementioned SIP requirement is a MAY requirement 15 and thus not necessarily supported by all SIP implementations.

One has to assume that the aforementioned policy may be implemented: Thereby, the E2ENP UA 128 would not notice the 20 dual seizure, except for an indication of an unsuccessful negotiation or re-negotiation phase. But one could also get into dual seizure troubles if the contention resolution policy is not supported by the underlying SIP UA 110 implementation.

25

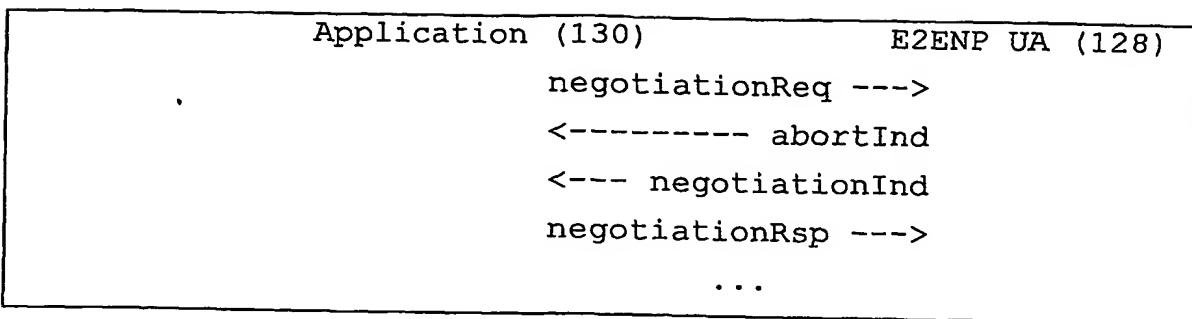
To tackle the latter case, the Cache 104 should store for each of its entries also the (E2ENPFromAddress, E2ENPTo-Address) couple associated with the outgoing INVITE and use this information for correlating an incoming INVITE with a 30 pending outgoing INVITE registered in the Cache 104, thus detecting the double seizure case: This would then trigger a contention resolution policy. This means that every incoming INVITE received after sending an INVITE should be checked by searching the Cache 104 with the aid of a secondary key, using the (From, To) couple. Although this is an extra effort,

it is only limited to the specific case of receiving an INVITE after having sent one.

As an alternative to the contention resolution policy suggested by [Ros1], which is eventually implemented by the underlying SIP UA 110, the E2ENP UA 128 shall support a more general policy, which is also applicable to complex scenarios like one-to-many or many-to-many described in [ReqSpec].

Whenever initiating a negotiation or pre-negotiation phase, the E2ENP UA 128 should not only include the session ID described in [ReqSpec] in the offer sent within the INVITE but also a hash thereof which takes the IP address of the computing unit where the E2ENP UA 128 is active into account and a time stamp. This hash might also be signed digitally in order to improve security aspects [ReqSpec]. The hash will be used as a substitute for the session ID in any subsequent SIP message. Should a dual seizure occur, the E2ENP would detect it by using the aforementioned (From, To) couple. Thereby, said contention resolution policy would be applied as follows:

- ♦ If the two peers are not already involved in multimedia sessions (which means no entry in the Cache 104 for a given local peer), each peer would compare the hash it generated with the one received from the peer. The peer with the biggest value is automatically chosen as Offerer and thus proceeds with the pre-negotiation or negotiation phase as such, whereas the other peer, the „loser“, automatically moves to the Answerer mode, according to the following MSC:



- ◆ If the two peers are already involved in multimedia sessions (which means an entry in the Cache 104 for the given (From, To) couple), the „bully“ process described in the previous point could be applied as well. The peer whose request is rejected could apply an unplanned re-negotiation after the winner of the „bully“ process has completed its negotiation. An example for this case is the simultaneous detection and re-negotiation reaction of the involved peers to a QoS violation.
- ◆ Since the generated session ID contains also a randomly generated number (together with the IP, port, etc. information) for the identification, it is not to be expected that some of the peers would always be „winners“ and some always „losers“.
- Finally, the aforementioned contention resolution process also applies to the case of a negotiation procedure colliding with a pre-negotiation procedure or even a lease-renewed procedure as well as for the collision of a pre-negotiation procedure with a lease-renewed procedure.

The model described in the previous subsections represents the static definition of the E2ENP UA FSM 106. At runtime, the E2ENP UA 128 shall associate a specific E2ENP session with an object representing it. This object - the Session - shall contain any information related with the given E2ENP

instance: most notably, a pointer to the current state and the instance of all the condition variables defined in the model above.

- 5    A thread of execution shall dynamically be associated with each of these Session objects: either created on demand, or - better - picked up from a thread-pool (potentially with lazy initialization).
- 10    ◆ This version of the E2ENP is based on the SIP protocol specification described in [Ros1]. If at later time another session level protocol or a more generic approach is chosen, the FSM 106 should be reworked in order to mask the corresponding changes from applications 130 or QoS Brokers 130, thanks to the E2ENP UA API 101b abstraction. This means that the interaction of the FSM 106 with the E2ENP UA API 101b Service User is a design-invariant.
- 15    ◆ The use of the SIP MESSAGE method is tentatively indicated in Fig. 30. Alternatively, the SIP OPTIONS method could be used. In the scope of the underlying invention, only the latter is exposed by the SIP UA Generic API 101c since the use of the former for implementing the FSM 106 model described below is still being investigated.
- 20    ◆ In order to avoid that entities interfacing with the E2ENP UA 128 (e.g. the E2ENP UA 128 Service Users, the SIP UA 110, and E2ENP management entities 102) directly invoke E2ENP UA 128 incoming primitives in response to E2ENP UA 128 outgoing primitives before the E2ENP UA 128 has ever had a chance to reach a stable state configuration, the E2ENP UA 128 is by design always loosely coupled with the aforementioned external entities. Another reason for this design choice is the possibility that otherwise there would be a non-null probability that the entities inter-

facing with the E2ENP UA 128 might block indefinitely a E2ENP UA thread.

- ◆ This means that the E2ENP UA 128 shall enforce one incoming message queue for the whole E2ENP UA FSM 106 as well as an outgoing message queue for each of the interfaces IF1, IF2, and IF3. The reason for the latter queues is due to the fact that the E2ENP UA 128, which is a software component, can not make any assumption on the way applications 130 would handle primitives thrown by the UA.  
5
- ◆ This approach is not necessary for IF4 and IF5 since the Parser 112 and the Factory 114 are designed as thread-safe, re-entrant libraries accessed by the E2ENP UA FSM  
106 via synchronous interfaces.  
15
- ◆ Concurrent instances of the E2ENP UA FSM 106 are handled with the aid of a thread-pool.

20 The E2ENP UA FSM 106 depends on the following components:

- ◆ the E2ENP UA API 101b, exposing the E2ENP functionality as implemented by the FSM 106,  
25
- ◆ the Parser API 101d, used for accessing services dealing with the parsing of session descriptions contained in received SIP messages,
- ◆ the Factory API 101e, used for accessing services dealing with the creation of session descriptions to be inserted  
30 in outgoing SIP messages,
- ◆ the SIP UA Generic API 101c, used for accessing SIP services,

- ◆ the E2ENP Management API 101a, used for accessing services dealing with the overall E2ENP UA 128 management functionality, and
- ◆ the Cache 104, used for storing and correlating identifiers of a given session.

The E2ENP UA FSM model is described in Figs. 30, 31, 32, 34, and 36 in terms of UML State Charts and can be implemented by using various well-known design patterns. In any case, as indicated in Fig. 1, the FSMs 106 are associated at runtime with instances of the applied framework or design pattern to implement the model. Each instance is associated with a specific SIP phase [ReqSpec], i.e. pre-negotiation, lease renewal, or negotiation. Re-negotiations run on the same instance as the underlying session established with the corresponding negotiation.

In order to handle the life cycle of an FSM 106 instance (creation, activation, deactivation, and destruction), an FSM instance factory 106 (see Fig. 1) is envisioned. This should be a singleton object creating instances of the FSM 106 either upon invocation of specific E2ENP UA API 101b primitives (negotiationReq, prenegotiationReq, renewLeaseReq) or upon invocation of specific SIP UA Generic API 101c primitives (connectionInd, optionsInd, messageInd).

Fig. 37 depicts the MSC for a simple pre-negotiation scenario, wherein the correlation between the E2ENP UA API 101b and the SIP UA Generic API 101c via the E2ENP UA FSM 106 is highlighted.

Fig. 38 depicts the MSC for a simple session establishment scenario, wherein the correlation between the E2ENP UA API 101b and the SIP UA Generic API 101c via the E2ENP UA FSM 106 is highlighted. In the presented scenario, the confirmation

of resource reservation completion from the Offerer side arrives before the same happens at Answerer side. Likewise, it is assumed that the resource reservation process succeeds in reserving exactly the amount of the originally requested resource.

In the following subsection, the E2ENP UA factory 108 shall briefly be described.

10 The E2ENP UA factory 108 is a management functionality modeled according to the „abstract factory“ design pattern described in [Gam] for instantiating a specific E2ENP UA 128 implementation using an implementation-independent interface.

15 This entity handles the instantiation of all the classes described in the previous sections required to create an instance of the E2ENP UA 128. More specifically, the E2ENP UA factory 108 creates the following entities:

20 ◆ an E2ENP UA FSM factory, which in turn creates the E2ENP UA FSM static structure,

◆ the FactoryFactory 120, which in turn creates the E2ENP Factory 114,

◆ the ParserFactory 118, which in turn creates the E2ENP Parser 112, and

25 ◆ the Cache 104.

Thereby, said E2ENP UA factory 108 depends on the following components:

30 ◆ the application 130 and/or middleware 130, which may directly call the E2ENP UA factory 108, alternatively a management entity 102.

- ◆ the E2ENP UA FSM 106, whose factory is created by the given concrete E2ENP UA factory implementation.
- ◆ the Factory 114 and Parser 112, whose factories are created by the given concrete E2ENP UA factory implementation.
- ◆ the Cache 104, which is created by the given concrete E2ENP UA factory implementation.

5 The E2ENP UA factory 108 has been designed by using the object-oriented paradigm. Thus, it is hereby described by using 10 the Unified Modeling Language (UML).

15 In the following, the E2ENP object configuration parameters shall briefly be summarized.

16 The E2ENP UA 128 and its sub-components (FSM 106, Parser 112, Factory 114, Cache 104) are configurable elements. The initial configuration should consider the specific features of the peer using the E2ENP UA 128.

20 The E2ENP UA 128 should be initialized with the following parameters:

- ◆ The E2ENP UA FSMs 106 are initiated with specific time-outs for the crucial states, where prompt decisions about the performance should be taken. The length of the time-outs depends on the specific underlying protocol of the E2ENP UA 128 and on the performance features of the respective peer. The time-outs of the different devices and networks may vary with a certain extent. The implementers 25 should consider these variations by defining the time-outs.
- ◆ The Cache 104 does not need any configuration parameters and is initiated by the E2ENP UA 128.

- ◆ The Parser 112 and the Factory 114 are respectively configured with the used underlying protocol type and their respective class name. They are initiated by means of the E2ENP UA 128.

5

The configuration of the E2ENP UA 128 and its respective components can be performed either manually with the aid of a Graphical User Interface (GUI) or can be preset with configuration parameters read from a configuration file.

10

Concerning the E2ENP UA FSM 106, the following parameters are configurable:

- ◆ Thread-pool configuration parameters:

15

- \* Maximum number of threads in the E2ENP thread-pool.
- \* Selector indicating whether the pool should have a lazy initialization (i.e. threads are created only when required).

20

- Selector indicating whether thread-pool optimization is required (only when lazy initialization is enforced). For optimization reasons, the pool might be reconfigured at runtime based on the actual pool usage.

25

⇒ Thread-pool optimization timer: Upon expiration of such timer, the optimization process should be applied (i.e. unused threads released).

30

- ◆ Timers (cf. Tab. 19).

- ◆ Mutex:

- \* Timer for detecting potential indefinite lock of the mutex, eventually with the aid of a low-priority instance of the FSM 106 (cf. TM in Tab. 19).

5        \* The number of attempts to get said mutex is limited by a maximum number (Max-Attempts).

The SIP UA implementation 110 shall allow configuring at least the following parameters:

10

- ◆ Underlying Session Protocol: This is the protocol used by E2ENP for carrying its meta-messages (SIP, RMI, socket). In future specifications of E2ENP, other session-layer protocols could be used (e.g. H323).

15

- ◆ Underlying Transport Protocol: The type of transport protocol to be used by SIP. In future specifications, this parameter might become a configuration parameter of the SIP UA 110 itself.

20        The following subsection provides information about the decision to make communication between the E2ENP UA 128 and the Parser 112 respectively the Factory 114 synchronous. In this connection, the E2ENP UA 128 shall support multiple concurrent instances of the E2ENP UA FSM 106. In order to avoid an unbounded usage of OS resources, the E2ENP UA 128 shall enforce a thread pool.

30        Communication between the E2ENP UA 128 and the Parser 112 respectively Factory 114 could also be asynchronous, e.g. implemented by an active request interface and a passive confirmation interface. This option is not used within the scope of the underlying invention because some gain in performance and flexibility is opposed by much more complicated state models for the FSMs 106 of the E2ENP UA 128.

One consequence of this decision is that the E2ENP UA 128 and the Parser 112 (also the Factory 114) communicate synchronously, thus having to share a common address space,  
5 which in most circumstances is a reasonable assumption. Another consequence of this decision is that both the Parser-Factory and the FactoryFactory have to be designed as thread-safe, re-entrant libraries.

10 Due to the discussion above, it is decided that communication between the E2ENP UA 128 and the Parser 112 as well as the Factory 114 is synchronous.

15 In conclusion, the main advantageous differences between the underlying invention and the state of the art shall be emphasized. Briefly summarized, the main benefits of the proposed approach consists in

- a QoS pre-negotiation with a lease-based management of pre-  
20 negotiated information, QoS negotiation, and QoS re-negotiation, and
- a resource reservation and/or release coordination offered by the E2ENP UA 128.

25 Moreover, the information to be negotiated (capabilities, application-level QoS Contracts, network-level QoS Contracts, stream-level adaptation paths, and stream-group-level adaptation paths) can be expressed in an interchangeable format in  
30 such a way that heterogeneous applications 130 and/or middleware 130 can easily agree on a reference model, which said applications 130 and/or said middleware 130 can then use for dynamically configuring Finite State Machines 106 (FSMs) in order to orchestrate local, peer, and network resources according to the preferences and profiles of the respective  
35

user, policies of the respective systems and network providers in a coordinated manner among a plurality of heterogeneous applications 130 and middleware 130 used by various peers.

5

Finally, said E2ENP UA 128 can be used for any session-layer protocol and session description language, a plurality of applications 130 and different types of middleware 130.

Table A: Used Abbreviations

Abbr.	Brief Explanation
ABNF	Augmented Backus-Naur Form
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
E2ENP	End-to-End Negotiation Protocol
FSM	Finite State Machine
IF<X>	Interface <X>
IP	Internet Protocol
IPv6	IP version 6
LAN	Local Area Network
MIME	Multi-Purpose Internet Mail Extensions
MSC	Message Sequence Chart (UML)
OSI	Open Systems Interconnection
QoS	Quality of Service
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAP	Service Access Point
SDP	Session Description Protocol
SDPng	Session Description Protocol (next generation)
SIP	Session Initiation Protocol
UML	Unified Modeling Language
XML	Extensible Markup Language

Table B: Depicted Features and their Corresponding Reference Signs

No.	Technical Feature
100	block diagram showing the applied architecture for the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP) according to the underlying invention
101a	E2ENP Management API (= interface IF1) between the E2ENP UA Factory 108 and the Management Entities 102
101b	E2ENP UA API (= interface IF2) between the E2ENP UA 128 and any application 130 or middleware 130 using the services offered by said E2ENP UA 128
101c	SIP UA Generic API (= interface IF3) between the E2ENP UA FSM 106 and the carrier protocol UA 110 (SIP, RMI, etc.)
101d	Parser API (= interface IF4) between the E2ENP UA FSM 106 and the implemented Parser 112
101e	Factory API (= interface IF5) between the E2ENP UA FSM 106 and the implemented Factory 114
102	E2ENP Management Entities of the E2ENP architecture 100
103	Cache API between the Cache 104 and the E2ENP UA FSM 106
104	two-stage Cache of the E2ENP architecture 100, connected to the Finite State Machine (FSM) 106 of the E2ENP User Agent (UA) 128
106	Finite State Machine (FSM) applied to the E2ENP User Agent (UA) 128 of the E2ENP architecture 100
108	E2ENP UA generation factory of the E2ENP architecture 100
108a	SIP-based E2ENP UA generation factory of the E2ENP architecture 100
110	placeholder for the implementation of the carrier protocol (SIP, RMI, etc.) User Agent (UA), also referred to as Service Provider

No.	Technical Feature
110a	SIP User Agent (UA) used for the UA implementation 110 of the carrier protocol within the applied E2ENP architecture 100 according to the first implementation example 200
110b	Java Remote Method Invocation (RMI)-based User Agent (UA) used for the UA implementation 110 of the carrier protocol within the applied E2ENP architecture 100 according to the second implementation example 300
110c	socket-based User Agent (UA) used for the UA implementation 110 of the carrier protocol within the applied E2ENP architecture 100 according to the third implementation example 400
112	placeholder for the implementation of the applied Parser
112a	SDPng Parser used for the Parser implementation 112 of the applied E2ENP architecture 100 according to the first implementation example 200
112b	Dummy Parser used for the Parser implementation 112 of the applied E2ENP architecture 100 according to the second implementation example 300. The Dummy Parser is used together with the Java RMI based UA 110b.
112c	Dummy Parser used for the Parser implementation 112 of the applied E2ENP architecture 100 according to the third implementation example 400. The Dummy Parser is used together with the socket based UA 110c.
114	placeholder for the implementation of the applied Factory
114a	SDPng Factory used for the Factory implementation 114 of the applied E2ENP architecture 100 according to the first implementation example 200
114b	Dummy Factory used for the Factory implementation 114 of the applied E2ENP architecture 100 according to the second implementation example 300. The Dummy Factory is used together with the Java RMI based UA 110b.

No.	Technical Feature
114c	Dummy Factory used for the Factory implementation 114 of the applied E2ENP architecture 100 according to the third implementation example 400. The Dummy Factory is used together with the socket based UA 110c.
116	Generation factory of the carrier protocol User Agent (UA) implementation
116a	Java-based SIP agent 110a generation factory for the carrier protocol User Agent 110 for SIP according to the first implementation example 200
116b	RMI agent 110b generation factory for the carrier protocol User Agent 110 for Java RMI according to the second implementation example 300
116c	Socket-based agent 110c generation factory for the carrier protocol User Agent 110 for socket-based connections according to the third implementation example 400
118	Generation factory of the applied Parser implementation 112
118a	SDPng Parser 112a generation factory (ParserFactory) of the applied Parser 112 according to the first implementation example 200
118b	Dummy Parser 112b generation factory (ParserFactory) of the applied Parser 112 according to the second implementation example 300
118c	Dummy Parser 112c generation factory (ParserFactory) of the applied Parser 112 according to the third implementation example 400
120	Generation factory of the applied Factory implementation 114
120a	SDPng Factory 114a generation factory (FactoryFactory) of the applied Factory 114 according to the first implementation example 200
120b	Dummy Factory 114b generation factory (FactoryFactory) of the applied Factory 114 according to the second im-

No.	Technical Feature
	plementation example 300
120c	Dummy Factory 114c generation factory (FactoryFactory) of the applied Factory 114 according to the third implementation example 400
122	Java-based SIP stack (JSIP) applied as a component of the said SIP User Agent (UA) 110a
124a	Remote Method Invocation (RMI) skeleton of the Java RMI-based User Agent (UA) 110b according to the second implementation example 300
124b	Remote Method Invocation (RMI) stub of the Java RMI-based User Agent (UA) 110b according to the second implementation example 300
126a	TX socket of the socket-based User Agent (UA) 110c according to the third implementation example 400 used for transmitting strings or objects
126b	RX socket of the socket-based User Agent (UA) 110c according to the third implementation example 400 used for receiving strings or objects
126c	Adapter of the socket-based User Agent (UA) 110c according to the third implementation example 400 used for receiving and/or transmitting strings or objects using the said sockets 126a+b
128	combined unit comprising said Cache 104, said E2ENP UA FSM 106, and said E2ENP UA Factory 108, in the following referred to as E2ENP-based User Agent (E2ENP UA)
130	multi-session multimedia application or QoS Broker (middleware) using the E2ENP UA API 101b to connect with the E2ENP UA 128
200	first implementation example of the applied architecture for the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP) according to one embodiment of the underlying invention using a Java-based SIP Stack (JSIP) and SDPng Parser and Factory

No.	Technical Feature
300	second implementation example of the applied architecture for the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP) according to one embodiment of the underlying invention using Sun's Java Remote Method Invocation (RMI)
400	third implementation example of the applied architecture for the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP) according to one embodiment of the underlying invention using the socket-based User Datagram Protocol (UDP) or Transmission Control Protocol (TCP)
500	UML class diagram showing the org::mind::e2enp package
502	E2ENPUserAgentListener interface within the org::mind::e2enp package
504	Provider interface within the org::mind::e2enp package
506	OffererListener interface, generalized to said interface Listener 502 within the org::mind::e2enp package
508	AnswererListener interface, generalized to said interface Listener 502 within the org::mind::e2enp package
510	ManagerProvider interface
512	ManagerListener interface
514	class ConfigurationRequest
516	class ParserFactoryConfiguration
600	Augmented Backus-Naur Form (ABNF) of the E2ENP address as a Universal Resource Identifier (URI)
700	first message sequence chart (MSC) showing the pre-negotiation procedure enabled by the User Agent (UA) 128 of the End-to-End Negotiation Protocol (E2ENP)
702	class OffererServiceUser, derived from the class E2ENPUserAgentListener, implementing the Listener interface 502
704	class OffererServiceProvider, derived from the class Provider, which implements the Provider interface 504
706	class AnswererServiceProvider, derived from the class

No.	Technical Feature
	Provider, which implements the Provider interface 504
708	class AnswererServiceUser, derived from the class E2ENPUserAgentListener, implementing the Listener interface 502
800	second message sequence chart (MSC) showing the session establishment with a QoS negotiation and resource reservation coordination enabled by the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)
900	UML class diagram showing the org::mind::e2enp::Cache sub-package
902	class LevelOneCache representing the first level (L1) of the two-stage Cache 104
904	class LevelTwoCache representing the second level (L2) of the two-stage Cache 104
906	class LevelOneEntry encompassing methods for storing and loading information to/from the first level (L1) of said two-stage Cache 104
908	class LevelTwoEntry encompassing methods for storing and loading information to/from the second level (L2) of said two-stage Cache 104
1000	diagram showing the top-level view of an Extensible Markup Language (XML) description used for the End-to-End Negotiation Protocol (E2ENP)
1001	Complex Type definition descType of the SDPng desc element. The illustration also incorporates the E2ENP specific changes made by the redefine mechanism.
1002	„e2enp:purpose“ child element of the „desc“ element.
1003	„sdpng:def“ child element of the „desc“ element.
1004	„sdpng:cfg“ child element of the „desc“ element.
1005	„sdpng:constraints“ child element of the „desc“ element.
1006	„sdpng:conf“ child element of the „desc“ element.
1100	diagram showing the XML substitution groups used for the End-to-End Negotiation Protocol (E2ENP)

No.	Technical Feature
1101	Complex Type definition descType of the SDPng desc element. The illustration also incorporates the E2ENP specific changes made by the redefine mechanism.
1102	„e2enp:purpose“ child element of the „desc“ element.
1103	„sdpng:def“ child element of the „desc“ element.
1104	„sdpng:cfg“ child element of the „desc“ element.
1105	„sdpng:constraints“ child element of the „desc“ element.
1106	„sdpng:conf“ child element of the „desc“ element.
1107	„sdpng:d“ element, head of the substitution group which defines the valid child elements of the „sdpng:def“ section.
1108	„audio:codec“ element, member of the „sdpng:d“ substitution group.
1109	„e2enp:qosdef“ element, member of the „sdpng:d“ substitution group.
1110	„rtp:pt“ element, member of the „sdpng:d“ substitution group.
1111	„rtp:transport“ element, member of the „sdpng:d“ substitution group. This element in turn is head of the „rtp:transport“ substitution group.
1112	„rtp:udp“ element, member of the „rtp:transport“ substitution group.
1113	„video:codec“ element, member of the „sdpng:d“ substitution group.
1200	diagram showing the XML purpose element used for the End-to-End Negotiation Protocol (E2ENP)
1201	„purpose“ element. Can be used to convey additional information about the description following in the protocol data unit (PDU).
1202	„e2enp:use“ element. Child of the „purpose“ section, can be used to indicate referenced sessions.
1203	„description“ element. Can be used to indicate a description PDU.

No.	Technical Feature
1204	„mediation“ element. Can be used to indicate a mediation PDU.
1205	„e2enp:session“ element
1206	„expires“ element
1300	diagram showing the XML qosdef element used for the End-to-End Negotiation Protocol (E2ENP)
1300'	XML qosdef element
1301	„audio:codec“ element. Describes an audio capability of the system.
1302	„video:codec“ element. Describes an video capability of the system.
1303	„rtp:pt“ element. Can be used to specify a desired payload format for a given capability.
1304	„e2enp:policy“ element. Can be used to specify the resource management policy to enforce.
1305	„e2enp:audio“ element. Can be used to describe the QoS contracts for audio streams.
1306	„e2enp:video“ element. Can be used to describe the QoS contracts for video streams.
1307	„e2enp:control“ element. Can be used to describe the QoS contracts for control streams.
1308	„e2enp:data“ element. Can be used to describe the QoS contracts for data streams.
1309	„rtp:map“ element. Can be used to define a mapping between an application level QoS Contract and a specific RTP payload format.
1310	„e2enp:contract“ element
1311	„predicate“ element
1312	„criterion“ element
1400'	XML qoscfg element
1400	diagram showing the XML qoscfg element used for the End-to-End Negotiation Protocol (E2ENP)
1401	„e2enp:context“ element. Can be used to define associa-

No.	Technical Feature
	tions between data streams and thus to express time synchronization and/or QoS correlation. Briefly, this element defines high level QoS Contracts.
1402	„e2enp:adopath“ element
1403	„default“ element
1404	„alt“ element
1405	„event“ element
1406	„path“ element
1407	„comp“ element
1408	„constraints“ element
1409	„par“ element
1500	UML message sequence chart (MSC) showing the interaction between the E2ENP User Agent (UA) 128 according to the underlying invention and the applied Parser 112 and Cache 104
1700	UML class diagram showing the general structure of a Document Object Model (DOM) tree
1702	class DocumentTree, which models the general structure of said DOM tree and can therefore be interpreted as the document root element
1704	class DocumentNode aggregated to said class DocumentTree 1702, which in turn aggregates between 0 and n children, thus recursively describing the tree structure
1800	UML class diagram showing a structural overview of the Parser implementation 112 using a „visitor design pattern“ for traversing the DOMTree 1804 and generating a XMLObject 1802
1802	class XML Object used within the Parser implementation 112
1804	class DOMTree, which models the general structure of said DOM tree and can therefore be interpreted as the document root element
1806	class DOMNode aggregated to said class DOMTree 1804

No.	Technical Feature
1808	generic Parser visitor class pVisitor, which visits between 1 and N DOMNodes
1810a	first specialized DOMNode of said class DOMNode 1806
1810n	N-th specialized DOMNode of said class DOMNode 1806
1812a	first specialized Parser visitor class pVisitor 1, which is applied to handle the first derived DOMNode
1812n	N-th specialized Parser visitor class pVisitor N, which is applied to handle the N-th derived DOMNode
1900	UML message sequence chart (MSC) showing the interaction between the E2ENP User Agent (UA) 128 according to the underlying invention and the applied Factory 114 and Cache 104
2000	UML class diagram showing a structural overview of the Factory implementation 114
2002	generic Factory visitor class fVisitor, which visits between 1 and N instances of ObjectGraphNode classes
2004	class ObjectGraphNode used within the Factory implementation 114
2006a	first specialized Factory visitor class fVisitor 1, which is applied to handle the first derived Node 2008a of the class ObjectGraphNode 2004
2006n	N-th specialized Factory visitor class fVisitor N, which is applied to handle the N-th derived Node 2008n of the class ObjectGraphNode 2004
2008a	first specialized Node (subclass) of the class ObjectGraphNode 2004
2008n	N-th specialized Node (subclass) of the class ObjectGraphNode 2004
2100	UML class diagram showing the org::mind::sip::sipApi package
2102	class management of the org::mind::sip::sipApi package
2104	class userAgent of the org::mind::sip::sipApi package
2106	class registrar of the org::mind::sip::sipApi package

No.	Technical Feature
2107	class time of the org::mind::sip::sipApi package
2108	interface SipListener of the org::mind::sip::sipApi package
2110	interface SipProvider of the org::mind::sip::sipApi package
2112	class SipExpiresHandling of the org::mind::sip::sipApi package
2114	class SipEndUser of the org::mind::sip::sipApi package
2200	UML class diagram showing the org::mind::sip::sipApi::userAgent package
2202	interface SipUserAgentListener of the org::mind::sip::sipApi::userAgent package
2204	interface SipUserAgentServerListener of the org::mind::sip::sipApi::userAgent package, derived from the class SipUserAgentListener 2202
2206	interface SipUserAgentClientListener of the org::mind::sip::sipApi::userAgent package, derived from the class SipUserAgentListener 2202
2208	Class SipConnectionReq of the org::mind::sip::sipApi::userAgent package
2210	interface SipUserAgentType of the org::mind::sip::sipApi::userAgent package
2212	interface SipUserAgentFactory of the org::mind::sip::sipApi::userAgent package
2214	class SipUserAgent of the org::mind::sip::sipApi::userAgent package, derived from the class SipUserAgentType 2210
2214a	class UAClient of the org::mind::sip::sipApi::userAgent package, derived from the class SipUserAgent 2214
2214b	class UAServer of the org::mind::sip::sipApi::userAgent package, derived from the class SipUserAgent 2214

No.	Technical Feature
2216	class SipNegotiationModel of the org::mind::sip::sipApi::userAgent package
2218	class SipRegistrationReq of the org::mind::sip::sipApi::userAgent package
2220	class SipRegistrationCfm of the org::mind::sip::sipApi::userAgent package
2300	UML class diagram showing the org::mind::sip::sipApi::registrar package
2302	interface SipRegistrarListener of the org::mind::sip::sipApi::registrar package
2304	interface SipRegistrarType of the org::mind::sip::sipApi::registrar package
2306	class SipRegistrar of the org::mind::sip::sipApi::registrar package, derived from said class Sip- RegistrarType 2304
2308	class SipRegistrationInd of the org::mind::sip::sipApi::registrar package
2310	class SipRegistrationRsp of the org::mind::sip::sipApi::registrar package
2312	interface SipRegistrarFactory of the org::mind::sip::sipApi::registrar package
2400	UML class diagram showing the org::mind::sip::sipApi::management package
2402	interface SipManagementListener of the org::mind::sip::sipApi::management package
2404	interface SipManager of the org::mind::sip::sipApi::management package
2406	class SipConfigurationReq of the org::mind::sip::sipApi::management package
2500	UML class diagram showing the org::mind::sip::sipApi::time package
2502	interface SipExpiresType of the org::mind::sip::sipApi::time package

No.	Technical Feature
2504	class SipParseException of the org::mind::sip::sipApi::time package
2506	class SipExpires of the org::mind::sip::sipApi::time package
2508	interface SipExpiresFactory of the org::mind::sip::sipApi::time package
2600	UML message sequence chart (MSC) showing the configuration of the Service Provider and the binding of the Service User with said Service Provider
2700	UML message sequence chart (MSC) showing the connection establishment between the User Agents of a client and a server
2800	UML message sequence chart (MSC) showing the OPTIONS method used for the End-to-End Negotiation Protocol (E2ENP)
2900	UML message sequence chart (MSC) showing the connection release between the User Agents of a client and a server
3000	first state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the root state
3100	second state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the NegOfferer sub-state
3200	third state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the NegAnswerer sub-state
3300	fourth state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the ReNegOfferer sub-state
3400	fifth state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the ReNegAnswerer sub-state
3500	sixth state transition diagram for the _ResvMtx Finite

No.	Technical Feature
	State Machine (FSM) for allowing multiple requests to seize the mutex in a coordinated manner, based on their priority
3600	seventh state transition diagram for a nested Finite State Machine (FSM) showing the mutex-related procedures executed in the WaitForCoordDone sub-state
3700	UML message sequence chart (MSC) showing the pre-negotiation procedure needed for the correlation of the Application Programming Interface (API) of the E2ENP User Agent (UA) and the generic Application Programming Interface (API) of the SIP User Agent (UA), thereby using the above-mentioned Finite State Machine (FSM) of the E2ENP User Agent (UA)
3702	class OffererSIPServiceUser, implementing the interface SipUserAgentClientListener 2206
3704	class OffererSIPServiceProvider, derived from the class SipUserAgent 2214
3706	class AnswererSIPServiceProvider, derived from the class SipUserAgent 2214
3708	class AnswererSIPServiceUser, implementing the interface SipUserAgentServerListener 2204
3800	UML message sequence chart (MSC) showing the session establishment with the QoS negotiation procedure and the resource reservation coordination needed for the correlation of the Application Programming Interface (API) of the E2ENP User Agent (UA) and the generic Application Programming Interface (API) of the SIP User Agent (UA), thereby using the above-mentioned Finite State Machine (FSM) of the E2ENP User Agent (UA)
3900	Tab. 1: table showing a list of primitives implemented by the Service Provider, based on a Java implementation of the Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation

No.	Technical Feature
	Protocol (E2ENP)
4000	Tab. 2: table showing a list of primitives to be implemented by the Service User, based on a Java implementation of the Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)
4100	Tab. 3: table showing a list of primitives to be implemented by the Service User acting as an Offerer, based on a Java implementation of the Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)
4200	Tab. 4: table showing a list of primitives to be implemented by the Service User acting as an Answerer, based on a Java implementation of the Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)
4300	Tab. 5: table showing the methods provided by the Application Programming Interface (API) of the first Cache level
4400	Tab. 6: table showing the methods provided by the Application Programming Interface (API) of the second Cache level
4500	Tab. 7: table showing a list of primitives which have to be implemented by the Application Programming Interface (API) of the Parser
4600	Tab. 8: table showing a list of primitives which have to be implemented for generating a specific parser instance
4700	Tab. 9: table showing a list of primitives which have to be implemented by the Application Programming Interface (API) of the Factory
4800	Tab. 10: table showing a list of primitives which have to be implemented for generating a specific factory instance

No.	Technical Feature
4900	Tab. 11: table showing the methods provided by the well-known factory-factory class E2ENPContentHandlerFactory
5000	Tab. 12: table showing a list of primitives implemented by the Service Provider, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)
5100	Tab. 13: table showing a list of client-side-specific primitives implemented by the Service User, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)
5200	Tab. 14: table showing a list of server-side-specific primitives implemented by the Service User, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)
5300	Tab. 15: table showing a list of both client-side- and server-side-specific primitives implemented by the Service User, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)
5400	Tab. 16: table showing a list of primitives implemented by the Service Provider, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)
5500	Tab. 17: table showing a list of primitives implemented by the Service User, based on a Java implementation of the generic Application Programming Interface (API) applied to the User Agent (UA) of the End-to-End Negotiation Protocol (E2ENP)

No.	Technical Feature
5600	Tab. 18: state transition table showing a survey of the applied exception events
5700	Tab. 19: table showing the applied timers applied to the End-to-End Negotiation Protocol (E2ENP) according to the underlying invention